Чаз Эмерик, Брайен Карпер, Кристоф Гранд

Программирование на Clojure

Chas Emerick, Brian Carper, Cristophe Grand

Clojure Programming

Practical Lisp for the Java World

O'REILLY®

Чаз Эмерик, Брайен Карпер, Кристоф Гранд

Программирование на Clojure

Практика применения Lisp в мире Java

ДМК ПРЕСС
издательство
Москва, 2015

УДК 004.432.42Clojure
ББК 32.973-018.1
Э54

Эмерик Ч., Карпер Б., Гранд К.

Э54 Программирование на Clojure: Пер. с англ. Киселева А. Н. – М.: ДМК Пресс, 2015. – 816 с.: ил.

ISBN 978-5-97060-299-7

Почему многие выбирают Clojure? Потому что это функциональный язык программирования, не только позволяющий использовать Java-библиотеки, службы и другие ресурсы JVM, но и соперничающий с другими динамическими языками, такими как Ruby и Python.

Эта книга продемонстрирует вам гибкость Clojure в решении типичных задач, таких как разработка веб-приложений и взаимодействие с базами данных. Вы быстро поймете, что этот язык помогает устранить ненужные сложности в своей практике и открывает новые пути решения сложных проблем, включая многопоточное программирование.

Издание предназначено для программистов, желающих освоить всю мощь и гибкость функционального программирования.

УДК 004.432.42Clojure
ББК 32.973-018.1

ISBN 978-1-449-39470-7 (анг.)

ISBN 978-5-97060-299-7 (рус.)

Содержание

Предисловие к русскому изданию

Вы держите в руках новую книгу, выпущенную издательством «ДМК Пресс», которая посвящена относительно молодому языку программирования Clojure, объединяющим мощь Lisp с популярностью платформы JVM, позволяя очень быстро создавать сложные приложения.

Долгое время на русском языке было доступно небольшое количество материалов, например, статья «Clojure, или “Вы все еще используете Java? Тогда мы идем к вам!”», опубликованная в четвертом номере журнала «Практика функционального программирования» (http://fprog.ru/). Но постепенно новый язык начал набирать популярность и среди русско-язычных разработчиков. Одно из свидетельств этого – на вводный курс о Clojure, организованный Дмитрием Бушенко (https://www.facebook.com/groups/clojure.course/), записалось почти 90 человек.

Данная книга написана разработчиками, давно работающими с Clojure и создавшими большое количество популярных библиотек. Книга описывает как сам язык, так и основные приемы программирования на нем, очень часто непривычные для людей, использовавших только императивные или объектно-ориентированные языки. Но непривычность языка не должна вас отпугивать, вы сможете использовать новые приемы не только при программировании на Clojure, но и в «стандартных» языках.

Помимо описания самого языка достаточно большая часть книги посвящена практическим аспектам его применения – веб-программированию, работе с базами данных, взаимодействию с кодом написанном на Java и т.д.

Если у вас возникнут вопросы по разработке на Clojure, вы можете задать их в группе ru_clojure в LiveJournal (http://ru-clojure.livejournal.com/) или в списке рассылки clojure-russian (https://groups.google.com/forum/?fromgroups#!forum/clojure-russian). Существует также аггрегатор русско-язычных блогов, посвященных Clojure (http://feeds.feedburner.com/RussianClojurePlanet), а если вы заинтересованы вопросами функционального программирования, то найдете много полезной информации в Russian Lambda Planet (http://fprog.ru/planet/).

Если вы захотите продолжить знакомство с языком, то можете получить больше информации в основном списке рассылки Clojure (https://groups.google.com/forum/?fromgroups#!forum/clojure), в постингах на Planet Clojure (http://planet.clojure.in) и других книгах, таких как «The Joy of Clojure».

Интересного чтения & happy hacking!

Алекс Отт

Благодарности

При подготовке перевода данной книги к печати ее рукописи были переданы для обсуждения членам русскоязычного сообщества пользователей Clojure, принявшим самое деятельное участие в ее улучшении.

Особую благодарность издательство выражает Дмитрию Бушенко, Сергею Париеву и Федору Русаку. Спасибо вам, друзья! Вашу помощь переоценить невозможно!

Предисловие

Clojure – динамический, строго типизированный язык программирования, основанный на виртуальной машине Java (Java Virtual Machine, JVM) и созданный уже пять лет тому назад. Он с восторгом был воспринят программистами, использующими самые разные языки и работающими в самых разных областях. Язык Clojure обладает весьма привлекательным набором возможностей и характеристик, приспособленных для решения современных задач программирования:

- поддержка функционального программирования, включая комплекс неизменяемых структур данных, по своей производительности приближающихся к обычным, изменяемым структурам;
- зрелая и эффективная среда выполнения, предоставляемая JVM;
- механизмы взаимодействий с JVM/Java, отвечающие самым широким архитектурным и эксплуатационным требованиям;
- комплекс надежных средств поддержки параллельного выполнения и семантики параллельного выполнения;
- будучи диалектом языка Lisp, предоставляет особенно гибкие и мощные средства метапрограммирования.

Язык Clojure представляет собой серьезную альтернативу для тех, кому постоянно приходится бороться с ограничениями обычных языков программирования и их окружений. Мы постараемся продемонстрировать это, показывая, насколько прозрачным выглядит взаимодействие Clojure с существующими технологиями, библиотеками и службами, используемыми программистами в повседневной работе. На протяжении всей книги мы будем знакомить вас с основами Clojure, опираясь в первую очередь на общественный опыт и знания, а не на (зачастую чужеродные) основные принципы информатики.

Кому адресована эта книга?

Мы писали эту книгу с расчетом на две категории специалистов. Надеемся, что вы принадлежите к одной из них.

Язык Clojure не только не отстает, но и часто превосходит многие основные языки программирования по выразительности, лаконичности и

гибкости, позволяя при этом пользоваться преимуществами высокой производительности, богатства библиотек, наличия обширного сообщества и устойчивости JVM. Все это делает его естественным шагом вперед для разработчиков программ на языке Java (и даже разработчиков для JVM, использующих интерпретируемые и не особенно быстрые языки программирования, отличные от Java), кого не устраивает низкая производительность или кто не может отказаться от своих инвестиций в JVM. Не менее естественным шагом вперед Clojure является также для разработчиков на Ruby и Python, которые не готовы пожертвовать выразительностью языка, но желающие получить более надежную и эффективную платформу выполнения и огромный выбор высококачественных библиотек.

Для Java-разработчиков

В мире живут и работают миллионы разработчиков на Java, но лишь немногие из них пишут программы для окружения, предъявляющего строгие требования, решая нетривиальные, часто узкоспециализированные задачи. Если вы относитесь к их числу, вероятно, вы постоянно находитесь в поисках более удобных инструментов и приемов, которые позволили бы повысить производительность труда, а также ценность вашего коллектива, организации или сообщества. Кроме того, вам, возможно, уже приходилось огорчаться, сталкиваясь с ограничениями в Java, отсутствующими в других языках, но при этом экосистема JVM не потеряла для вас своей привлекательности: трудно отказаться от зрелой среды выполнения, богатого выбора сторонних библиотек, поддержки поставщиками и огромного накопленного опыта, независимо от того, какими яркими не выглядели бы перспективы использования альтернативных языков.

В Clojure вы найдете долгожданное облегчение. Программы на этом языке выполняются под управлением JVM и обладают превосходной производительностью. Этот язык позволяет использовать любые имеющиеся библиотеки, инструменты и приложения. Он *проще* чем Java, лаконичнее и при этом намного выразительнее.

Для разработчиков на Ruby, Python и других языках

Языки Ruby и Python далеко не новые, но в последние годы они приобрели особую популярность (можно даже сказать: «заняли господствующее положение»). Не трудно понять почему: оба являются выразительными, динамическими языками, поддерживаются бурно разрастающимися сообществами, обеспечивают высокую производительность труда во многих сферах.

Для вас язык Clojure также является естественным выбором. Как программист на Ruby или Python вы, вероятно, не желаете терять мощь этих языков, но вы можете испытывать потребность в более надежной платформе, обла-

дающей более высокой производительностью и богатым выбором библиотек. Язык Clojure, основанный на JVM, целиком и полностью отвечает этим требованиям – он полностью соответствует, а иногда и превосходит другие языки по своей выразительности и способности повышать производительность труда разработчика.

Примечание. Мы часто будем сравнивать Clojure с Java, Ruby и Python, чтобы помочь вам перенести свой опыт на Clojure. В таких сравнениях мы всегда будем подразумевать канонические реализации этих других языков:
- Ruby MRI (также называется, как CRuby);
- CPython;
- Java 6/7.

Как читать эту книгу

Работая над этой книгой, мы хотели как можно больше наполнить ее конкретными, подробными и практическими примерами, достаточно понятными, чтобы вы могли избежать ошибок. Мы не раз сталкивались с книгами, где от начала до конца рассматривался пример разработки единственной программы или приложения. Такой подход, на наш взгляд, разрушает целостность повествования и вынуждает авторов от главы к главе исследовать вымученный «практический» пример, который может не соответствовать кругу задач, решаемых читателем.

Учитывая это, мы разделили книгу на две основные части, и одну вступительную, где излагаются основы языка, занимающими примерно две трети книги. За ними следует четвертая часть с большим количеством практических примеров из разных прикладных областей. Такое деление на части с совершенно разным содержимым позволяет квалифицировать эту книгу, как «книгу двойного назначения». (Автором этого термина, возможно, является Мартин Фаулер (Martin Fowler), употребивший его в статье http://martinfowler.com/bliki/DuplexBook.html.) В любом случае, мы предполагаем два подхода к чтению этой книги.

Начните с практического применения Clojure

Часто лучший способ изучить язык заключается в том, чтобы применить его на практике. Если такой подход кажется вам более привлекательным, есть шанс, что вы найдете в книге пару практических примеров, демонстрирующих решение задач, с которыми вам приходится сталкиваться ежедневно, благодаря которым вы сможете провести параллели в решении некоторых категорий задач на используемом вами языке (или языках) и на языке Clojure. В этих примерах вы можете столкнуться с большим количеством не известных вам понятий и языковых конструкций. В подобных ситуациях, для по-

нимания новых концепций, используйте в качестве отправной точки контекст предметной области и учебный материал в первой части книги.

Начните с последовательного изучения основ языка Clojure

Иногда, чтобы по-настоящему разобраться в чем-то, необходимо досконально изучить внутреннее устройство, начиная с самых основ. Если вы предпочитаете такой подход, тогда лучше будет начать «переваривать» эту книгу с самого начала, с первой страницы в главе 1. Мы постарались последовательно и подробно объяснить все основополагающие принципы и конструкции языка Clojure, поэтому вам редко придется заглядывать вперед, чтобы понять концепции, встречающиеся раньше их описания. Выбирая подход последовательного освоения основ языка Clojure, не стесняйтесь забегать вперед и заглядывать в практическую часть книги, где вы найдете интересные примеры, схожие с задачами, решаемыми вами.

Кто мы?

Мы – три разработчика программного обеспечения (ПО), разными путями пришедшие в Clojure и осознавшие его ценность. В процессе работы над книгой мы старались вложить в нее все наши представления о том, как и почему следует использовать Clojure, чтобы вы с успехом смогли применять его на практике.

Чаз Эмерик

Чаз Эмерик (Chas Emerick) стал постоянным членом сообщества Clojure в начале 2008. Занимался разработкой ядра языка, принимал участие в десятках проектов с открытым исходным кодом на языке Clojure, а также часто выступал и писал статьи о языке Clojure и разработке ПО в целом.

Чаз занимается сопровождением проекта Clojure Atlas (http://clojureatlas.com) визуализации языка Clojure и его стандартной библиотеки с целью обучения.

Является основателем Snowtide (http://snowtide.com), небольшой компании в Западном Массачусетсе, занимающейся производством ПО. Основной круг интересов Чаза лежит в области извлечения неструктурированных данных с уклоном в обработку документов PDF. Он пишет статьи и книги о Clojure занимается разработкой ПО и предпринимательством, а также имеет другие пристрастия (http://cemerick.com).

Брайен Карпер

Брайен Карпер (Brian Carper) – программист на Ruby, ставший приверженцем Clojure. Занимается программированием на Clojure, начиная с 2008,

и использует этот язык и дома, и на работе для всего подряд, от разработки веб-приложений до анализа данных в приложениях с графическим интерфейсом.

Брайен является автором приложения Gaka (https://github.com/briancarper/gaka), компилятора Clojure-to-CSS, и библиотеки объектно-реляционного отображения Oyako (https://github.com/briancarper/oyako). Пишет статьи о Clojure и на другие темы по адресу: http://briancarper.net.

Кристоф Гранд

Кристоф Гранд (Christophe Grand) – давний поклонник функционального программирования, заплутавший на просторах Java, пока в начале 2008 не встретил Clojure и не влюбился в него с первого взгляда! Является автором Enlive (http://github.com/cgrand/enlive), библиотеки управления шаблонами, обеспечивающей возможность преобразования и извлечения HTML/XML; Parsley (http://github.com/cgrand/parsley), инкрементального парсер-генератора; и Moustache (http://github.com/cgrand/moustache), веб-фреймворка, включающего поддержку маршрутизации и промежуточных функций для Ring.

Как независимый консультант, занимается разработкой ПО и предлагает обучение языку Clojure. Он также пишет статьи о Clojure на сайте http://clj-me.cgrand.net.

Благодарности

Как любой значительный труд, эта книга была бы невозможна без усилий десятков, если не сотен людей.

В числе первых хотелось бы назвать Ричарда Хикки (Rich Hickey), создателя языка Clojure. Всего за несколько коротких лет он спроектировал, реализовал и вывел в свет новый язык программирования, который для многих стал не просто еще одним инструментом, но и вдохнул новую влюбленность в программирование. Кроме того, он лично многому научил нас, не только программированию, но также терпению, скромности и видению перспектив. Спасибо, Рич!

Дейв Файрам (Dave Fayram) и Майк Лукидис (Mike Loukides) оказали существенную помощь в выработке первначальной концепции и организации книги. Конечно, вы сейчас не держали бы эту книгу в руках, если бы не наш редактор Джулия Стил (Julie Steele) и остальные сотрудники O'Reilly, занимавшиеся подготовкой книги к публикации.

Качество книги было бы намного ниже, если бы не наши технические редакторы, в число которых вошли Сэм Аарон (Sam Aaron), Энтони Батчелли (Antoni Batchelli), Том Фоулхабер (Tom Faulhaber), Крис Грангер (Chris Granger), Энтони Гримс (Anthony Grimes), Фил Хагельберг (Phil Hagelberg), Том Хикс (Tom Hicks), Алекс Миллер (Alex Miller), Вильям Морган (William Morgan), Лоурент Петит (Laurent Petit) и Дин Вамплер

(Dean Wampler). Мы также выражаем признательность всем, кто присылал свои отзывы и комментарии к черновикам книги на форумах издательства O'Reilly, по электронной почте, в твиттере и так далее.

Майкл Фогус (Michael Fogus) и Крис Хаузер (Chris Houser) многократно становились для нас источником вдохновения. Например, представив способы взаимодействия с REPL в своей книге о Clojure «The Joy of Clojure», которые мы бессовестно скопировали и повторили.

Если мы кого-то забыли упомянуть, примите нашу благодарность и извинения; закончив этот тяжелый труд, мы хотели бы оставаться честными и последовательными!

И последнее, но не в последнюю очередь

Сообщество Clojure оставалось моим вторым домом на протяжении ряда лет. Программисты на Clojure, отличающиеся приветливостью и положительной энергетикой, продолжают вдохновлять меня и служить примером для подражания. Например, многие завсегдатаи канала `#clojure` на сайте Freenode IRC, ставшие моими друзьями, помогли мне освоить многое из того, в чем без их помощи разобраться было бы очень трудно.

Моим соавторам, Кристофу и Брайену: работать с вами было честью для меня. Я не представляю, как можно было бы закончить этот труд без вашего участия.

Моим родителям, Чарли (Charley) и Дарлин (Darleen): мое неутолимое любопытство к внутреннему устройству вещей, моя любовь к языку и риторике и мои интересы в бизнесе – все это было заложено вами много лет тому назад. Без вашего влияния я определенно не смог бы найти свой путь в жизни, основать свою компанию или написать эту книгу – всего этого я достиг, благодаря настойчивости, воспитанной вами.

Наконец, моей супруге Крисси (Krissy): жертвы, которые ты принесла, чтобы позволить мне следовать своим амбициям, безмерны. Боюсь, что никогда не смогу в полной мере отблагодарить тебя за это. Поэтому я скажу просто: я люблю тебя.

– Чаз Эмерик (Chas Emerick),
февраль 2012

Всем членам сообщества, помогавшим в создании Clojure: спасибо за ваш беспримерный труд, за то что сделали мою профессиональную и личную жизнь намного более приятной и за то, что открыли мне глаза на новые перспективы.

Моим соавторам, Кристофу и Чазу: никогда прежде мне не доводилось работать со столь умными людьми. Для меня это было честью и привилегией.

Моей жене Николь (Nicole): прости, что не давал тебе заснуть по ночам треском клавиатуры.

– Брайен Карпер (Brian Carper),
февраль 2012

Ричарду Хикки (Rich Hickey), создавшему Clojure и сплотившему такое дружное сообщество.

Сообществу, приведшему меня к более высоким стандартам.

Моим соавторам, Брайену и Чазу: для было большой честью работать с вами.

Моему профессору Даниелю Гоффинье (Daniel Goffinet), чей острый ум радикально изменил мои взгляды на программирование и информатику в целом – этим я в значительной мере обязан ему, как никому другому.

Моим родителям: за вашу любовь и первый 8-разрядный компьютер, за которым я проводил слишком много времени, вызывая у вас беспокойство.

Моей супруге Эмили (Emilie) и моему сыну Гэлу (Gaël): спасибо вам за поддержку, которую вы оказывали, пока я работал над этой книгой.

– Кристоф Гранд (Christophe Grand),
февраль 2012

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив
: Используется для обозначения новых терминов, имен файлов и расширений имен файлов.

`Моноширинный шрифт`
: Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, типов данных, переменных окружения, инструкций и ключевых слов.

```
; строки в листингах, начинающиеся с точки с запятой
```
: Обозначают вывод (в стандартные потоки вывода и ошибок), полученный в результате выполнения программного кода в интерактивной оболочке REPL.

```
;= строки в листингах, начинающиеся с точки с запятой и знака равенства
```
: Обозначают результат/возвращаемое значение, полученный в ходе вычислений в интерактивной оболочке REPL.

`Моноширинный жирный`
: Обозначает команды или другой текст, который должен вводиться пользователем.

`Моноширинный курсив`
: Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Примечание. Так обозначаются советы, предложения и примечания общего характера.

Внимание. Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Clojure Programming by Chas Emerick, Brian Carper, and Christophe Grand (O'Reilly). Copyright 2012 Chas Emerick, Brian Carper, and Christophe Grand, 978-1-449-39470-7».

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

Safari® Books Online

Safari Books Online – это виртуальная библиотека, которая позволяет легко и быстро находить ответы на вопросы среди более чем 7500 технических и справочных изданий и видеороликов.

Подписавшись на услугу, вы сможете загружать любые страницы из книг и просматривать любые видеоролики из нашей библиотеки. Читать книги на своих мобильных устройствах и сотовых телефонах. Получать доступ к новинкам еще до того, как они выйдут из печати Читать рукописи, находящиеся в работе и посылать свои отзывы авторам. Копировать и вставлять отрывки программного кода, определять свои предпочтения, загружать отдельные главы, устанавливать закладки на ключевые разделы, оставлять примечания, печатать страницы и пользоваться массой других преимуществ, позволяющих экономить ваше время.

Благодаря усилиям O'Reilly Media, данная книга также доступна через услугу Safari Books Online. Чтобы получить полный доступ к электронной версии этой книги, а также книг с похожими темами издательства

O'Reilly и других издательств, подпишитесь бесплатно по адресу http://my.safaribooksonline.com.

Как с нами связаться

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

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

http://shop.oreilly.com/product/0636920013754.do

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте: http://www.oreilly.com.
Ищите нас на Facebook: http://facebook.com/oreilly.
Следуйте за нами в Твиттере: http://twitter.com/oreillymedia.
Смотрите нас на YouTube: http://www.youtube.com/oreillymedia.

Глава 1. Вниз по кроличьей норе

Если вы читаете эту книгу, значит вы открыты для изучения новых языков программирования. Мы также предполагаем, что вы ожидаете получить определенную выгоду от приложенных усилий, например, повышение эффективности вашего труда, увеличение продуктивности вашей команды и компании в целом.

Мы верим, что вам удастся войти в этот прекрасный цикл, когда вы изучаете, применяете и видите результат использования Clojure. Есть хорошая фраза: *язык Clojure требует изменить взгляды на разработку и в итоге это окупится*.

У нас, разработчиков, часто формируются очень запутанные и личные отношения со средствами разработки. Выбор инструментов обычно диктуется прагматизмом и наличием унаследованного кода. Но при прочих равных программисты выбирают средства, максимально эффективные и позволяющих раскрыть весь их потенциал в создании полезных и аккуратных систем. Как говорится, нам нужно то, что делает решение простых задач легким, а сложных – возможным.

Почему Clojure?

Clojure – язык программирования, соответствующий этому стандарту. Заимствовавший самое лучшее из нескольких языков программирования – включая различные реализации Lisp, а также Ruby, Python, Java, Haskell и других – Clojure обладает множеством особенностей, пригодных для решения самых сложных задач, с которыми приходится сталкиваться программистам в наши дни, и которые уже маячат на горизонте. А отсутствие необходимости переходить на совершенно новую, незнакомую платформу и среду выполнения (что обычно присуще многим другим языкам, появившимся в последние годы), так как Clojure выполняется под управлением виртуальной машины Java, нередко перевешивает все прагматические доводы и проблему наличия унаследованного кода в пользу выбора нового языка программирования.

Чтобы подогреть ваш интерес, перечислим некоторые из особенно примечательных особенностей языка Clojure:

Clojure выполняется под управлением JVM

В программах на языке Clojure можно использовать любые Java-библиотеки. Библиотеки на языке Clojure, в свою очередь, можно использовать в программах на Java. Приложения на языке Clojure могут упаковываться подобно любым Java-приложениям и разворачиваться везде, где могут разворачиваться другие приложения на Java: на серверах веб-приложений; на персональных компьютерах с интерфейсом Swing, SWT или командной строки; и так далее. Это также означает, что среда выполнения Clojure также является средой выполнения Java, одной из самых эффективных и надежных в мире.

Clojure – это Lisp

В отличие от Java, Python, Ruby, C++ и других членов семейства Algol-подобных языков программирования, Clojure принадлежит к семейству Lisp. Однако, забудьте все, что вы знаете (включая слухи) о Lisp: язык Clojure унаследовал все самое лучшее от Lisp, но не обременен недостатками и анахронизмами многих других реализаций Lisp. Кроме того, будучи диалектом Lisp, Clojure поддерживает макросы, обеспечивающие возможность метапрограммирования и расширения синтаксиса, ставшие за последние десятилетия эталоном, на который равняются другие подобные системы.

Clojure – язык функционального программирования

Язык Clojure поощряет использование функций, как объектов первого рода, а также функции высшего порядка, и предлагает собственный набор эффективных неизменяемых типов данных. Clojure фокусируется на функциональной парадигме, что устраняет типичные ошибки и недостатки, связанные с беспрепятственным изменением значений переменных. Кроме того, функциональный стиль значительно упрощает разработку параллельных программ.

Переменные в языке Clojure не являются переменными в классическом понимании

Прежде чем продолжить знакомство с языком программирования Clojure, необходимо внести дополнительную ясность, касающуюся термина «переменные».

В языке Clojure отсутствуют переменные в том понимании, в каком они существуют в императивных языках программирования, таких как C/C++, Java или Python. Их место занимают «переменные», своим поведением больше напоминающие константы.

Переменная в Clojure – это единовременная привязка значения к его имени (если точнее – к «символу» Clojure). Однажды привязав, скажем, значение 1 к символу `a`, изменить эту привязку, назначив значение 2 символу `a`, уже нельзя.
Тем не менее, в Clojure существуют специальные типы данных, позволяющие изменять значения внутри себя. Например, привязав объект типа `ref` к символу `b`, мы не сможем в дальнейшем изменить эту привязку. Но сам объект типа `ref` сможет изменять свое содержимое. Изменение (мутация) данных внутри таких объектов может происходить только в транзакциях. Как правило, начинающие Clojure-программисты часто используют этот механизм и применяют много изменяемых данных. Но настоящий путь Clojure состоит в максимально полном отказе от приемов, основанных на изменении переменных, настолько, насколько это вообще возможно.
Для обозначения описанных выше неизменяемых переменных создателями языка Clojure был даже придуман специальный термин: «var» (вероятно сокращение от «variable» – переменная), не имеющий аналогов в русском языке. По этой причине далее в книге под термином «переменная» будут подразумеваться переменные Clojure (vars), а для обозначения классических переменных будет использоваться термин «изменяемая переменная».

Clojure предлагает современные решения проблем, свойственных выполнению в многопоточной среде

Для эффективного использования многоядерных процессоров, многопроцессорных систем и систем распределенных вычислений нужно использовать языки и библиотеки, разрабатывавшиеся с учетом новых возможностей. Ссылочные типы в языке Clojure обеспечивают четкое разделение *состояния* (state) и *идентичности* (identity). Данная семантика упрощает реализацию параллельных вычислений, избавляя от необходимости использовать ручное управление блокировками, подобно тому как автоматическая сборка мусора облегчает управление памятью.

Clojure – динамический язык программирования

Clojure – динамический и строго типизированный язык программирования (и этим он напоминает языки Python и Ruby), однако вызовы функций компилируются в (быстрые!) вызовы Java-методов. Clojure является динамичным еще и в том смысле, что поддерживает возможность обновления и загрузки нового программного кода, локального или удаленного, прямо во время выполнения. Это особенно удобно для интерактивной разработки и отладки или даже для расширения и исправления удаленных приложений без их остановки.

Мы не ждем, что вы поняли все, о чем говорилось выше, но надеемся, что вы получили общее представление и оно вам понравилось.

Если так, поехали дальше. После прочтения этой главы вы сможете писать простые программы на Clojure и начнете понимать, как этот язык может помочь вам в ваших проектах.

Как получить Clojure

Чтобы опробовать программный код в этой главе и заняться исследованием языка Clojure, вам потребуются следующие два программных компонента:

1. Среда выполнения Java. Виртуальную машину Java, Oracle JVM, в версиях для Windows и Linux, можно загрузить бесплатно (http://java.com/en/download/). Все версии Mac OS X включают предустановленную виртуальную машину Java. Для Clojure требуется Java v1.5 или выше[1]; предпочтительнее использовать более современные версии – v1.6 или v1.7.
2. Сам язык Clojure доступен на clojure.org (http://clojure.org/downloads). *Весь программный код в этой книге требует версию v1.3.0 или выше, и был протестирован с версией v1.4.0*[2]. Внутри загруженного zip-архива можно найти файл с таким именем, как *clojure-1.4.0.jar*; это все, что вам потребуется для начала.

Примечание. Существует множество расширений, упрощающих разработку на языке Clojure в популярных средах разработки, таких как Eclipse и Emacs; см. раздел «Инструментарий» в главе 10, где приводится обзор инструментов поддержки Clojure. Для первых шагов на пути к пониманию Clojure вполне достаточно будет интерактивной командной оболочки REPL, тем не менее, мы рекомендуем вам пользоваться своим любимым текстовым редактором или интегрированной средой разработки (IDE), если в них есть поддержка Clojure, или найти соответствующий редактор.

Если вы затрудняетесь в выборе редактора или IDE, рекомендуем обратить внимание на Leiningen, наиболее популярный инструмент управления проектами на языке Clojure. Он автоматически загрузит Clojure, предоставит более удобную, по сравнению со стандартной, интерактивную оболочку REPL, и вы наверняка будете использовать его в повседневной работе, по крайней мере, первое время. Краткое описание инструмента приводится в разделе «Leiningen», в главе 8.

---

[1] Для Clojure 1.5 требуется версия Java, не ниже 1.6. – *Прим. ред.*

[2] Оценивая уровень поддержки обратной совместимости, установившийся за недолгую историю развития языка, можно утверждать, что программный код в книге будет выполняться и с более поздними версиями Clojure.

Если вы не желаете загружать что-либо прямо сейчас, многие примеры из этой книги вы сможете опробовать с помощью браузера, посетив страницу http://tryclj.com.

Интерактивная оболочка REPL для Clojure

Реализации многих языков включают интерактивную командную оболочку REPL, которую часто называют интерпретатором: в Ruby имеется `irb`; в Python имеется свой интерпретатор командной строки; в Groovy имеется консоль; даже в Java имеется нечто похожее на REPL в BeanShell. Аббревиатура «REPL» получена из простого описания принципа функционирования оболочки:

1. Read (прочитать): выполняется чтение программного кода из некоторого источника (обычно `stdin`, но может быть иной, например при использовании REPL в IDE).
2. Eval (вычислить): программный код выполняется, возвращая некоторое значение.
3. Print (вывести): значение передается в некоторый поток вывода (обычно `stdout`, но перед результатом, самим программным кодом может быть выведена дополнительная информация).
4. Loop (повторить): управление возвращается к шагу чтения (Read).

В языке Clojure также имеется интерактивная оболочка REPL. Но она отличается от аналогичных оболочек во многих других языках, потому что это не интерпретатор для ограниченного набора команд языка Clojure. Весь программный код, который вводится в Clojure REPL, *компилируется* в байт-код JVM, как если бы он был загружен из файла с исходным кодом. В обоих случаях компиляция производится во время выполнения, не требуя выделения отдельного этапа «компиляции»[1]. В действительности программный код Clojure *никогда* не интерпретируется. Эта особенность имеет два следствия:

1. Операции выполняются в REPL на «полной скорости»; нет никаких различий в скорости или семантике между кодом, выполняемым в REPL и в обычном приложении.

---

[1] При необходимости программный код на языке Clojure можно скомпилировать заранее. Подробности см. в разделе «Предварительная компиляция» в главе 8.

2. Поняв, как действует интерактивная оболочка REPL в языке Clojure (в особенности фазы чтения и выполнения), вы без труда поймете, как действует сам язык Clojure на самом базовом уровне.

А теперь, помня о втором пункте, погрузимся в Clojure REPL и посмотрим, сможем ли мы постичь основы.

> **Примечание.** Эффективная разработка на Clojure подразумевает более частое использование REPL, если сравнивать с другими языками. Такой подход делает разработку максимально интерактивной. Это не просто инструмент, помогающий писать код быстрее, он позволяет получить настоящее удовольствие от разработки на Clojure. Подробнее об этом мы поговорим в главе 10.

Пример 1.1. Запуск Clojure REPL из командной строки

```
% java -cp clojure-1.4.0.jar clojure.main
Clojure 1.4.0
user=>
```

Эта магическая команда начнет новый процесс JVM с *classpath*[1], включающим файл *clojure.jar* в текущем каталоге, и в качестве точки входа в программу будет использовать класс `clojure.main`[2]. Если вы еще не знаете, что такое classpath, загляните в раздел «Знакомство с classpath» в главе 8. А пока просто представляйте ее как близкий аналог переменной окружения `PYTHONPATH` в Python, `$:` в Ruby или `PATH` в командной оболочке операционной системы, то есть множество файлов и каталогов, откуда JVM будет загружать классы и ресурсы.

Когда на экране появится приглашение к вводу `user=>`, оболочка REPL будет готова к вводу программного кода на Clojure. Часть строки приглашения в Clojure REPL, предшествующая `=>`, является именем *текущего пространства имен*. Пространства имен похожи на модули или пакеты; мы подробно будем обсуждать их далее в этой

[1] Здесь под *classpath* подразумевается список каталогов для поиска классов, полученный из переменной окружения `CLASSPATH` или переданный виртуальной машине JVM в виде значения ключа `--classpath`. – *Прим. перев.*

[2] Также можно было бы выполнить команду `java -jar clojure.jar`, но флаг `-cp` и точка входа `clojure.main` играют важную роль, чтобы помнить о них; подробнее об этом будет рассказываться в главе 8.

главе, в разделе «Пространства имен». Сеанс работы Clojure REPL всегда начинается в базовом пространстве имен `user`.

Рассмотрим небольшой фрагмент кода – функцию, вычисляющую среднее арифметическое значение нескольких чисел, написанную на языках Java, Ruby и Python:

Пример 1.2. Вычисление среднего арифметического значения в Java, Ruby и Python

```
public static double average (double[] numbers) {
     double sum = 0;
     for (int i = 0; i < numbers.length; i++) {
          sum += numbers[i];
     }
     return sum / numbers.length;
}

def average (numbers)
     numbers.inject(:+) / numbers.length
end

def average (numbers):
     return sum(numbers) / len(numbers)
```

Ниже приводится эквивалентная функция на языке Clojure:

```
(defn average                                    ❶
     [numbers]                                   ❷
     (/ (apply + numbers) (count numbers)))      ❸
```

❶ Ключевое слово `defn` определяет в текущем пространстве имен новую функцию с именем `average`.

❷ Функция `average` принимает один аргумент, на который в теле функции можно ссылаться по имени `numbers`. Обратите внимание на отсутствие объявления типа – эта функция одинаково хорошо может обрабатывать любые коллекции или массивы чисел любых типов.

❸ В теле функции `average` вычисляется сумма переданных ей чисел (`apply + numbers`)[1], затем сумма делится на количество чисел (`count numbers`) и результат деления возвращается.

[1] Обратите внимание, что символ «плюс» (+) не является специальным оператором, как в большинстве других языков программирования. Это обычная функция, ничем не отличающаяся от других, определяемых нами функций. `apply` – еще одна функция, которая применяет переданную ей функцию к коллекции аргументов (в данном случае `numbers`); то есть, (`apply + [a b c]`) вернет то же значение, что и (`+ [a b c]`).

Выражение `defn` можно ввести в интерактивной оболочке REPL, затем вызвать функцию, передав ей вектор чисел, и получить ожидаемый результат:

```
user=> (defn average
         [numbers]
         (/ (apply + numbers) (count numbers)))
#'user/average
user=> (average [60 80 100 400])
160
```

Об оформлении примеров взаимодействия с REPL

С этого момента примем простое соглашение по оформлению листингов, демонстрирующих взаимодействие с оболочкой REPL, чтобы вы могли отличать разновидности вывода в ней. Значения, возвращаемые выражениями, будут предваряться префиксом `;=`:

```
(average [60 80 100 400])
;= 160
```

Содержимое, выводимое в `stdout` выражением, как противоположность автоматическому выводу значений, возвращаемых выражениями, будет предваряться единственной точкой с запятой:

```
(println (average [60 80 100 400]))
; 160
;= nil
```

Здесь показаны две строки с префиксами, потому что `println` возвращает `nil` после вывода значения или значений в `stdout`.

Строки, начинающиеся с точки с запятой, в языке Clojure являются комментариями, поэтому приводимые в книге примеры можно копировать целиком и вставлять в окно REPL, не заботясь об удалении лишних строк. Мы не включаем в листинги строку приглашения к вводу с именем пространства имен `namespace=>`, так как она не является допустимым программным кодом на языке Clojure и будет вызывать ошибку, если по неосторожности окажется скопированной в REPL.

Вам не придется путаться в частоколе скобок

Многих программистов, еще не пользовавшихся языком Lisp или с ностальгией вспоминающих студенческие времена, как изучали его в студенческие годы, берет оторопь при виде синтаксиса Lisp. Обычно такая реакция вызвана следующими причинами.

1. Использование круглых скобок для ограничения областей видимости, вместо более привычных фигурных скобок `{...}` или блоков `do ... end`.
2. Использование префиксной нотации для записи операций, например: `(+ 1 2)` вместо более привычной инфиксной формы `1 + 2`.

Такое отторжение обусловлено, прежде всего, непривычностью. Фигурные скобки, используемые для ограничения областей видимости в Java (C, C++, C#, PHP, ...), выглядят вполне привычно, зачем тогда использовать то, что выглядит запутанно? Аналогично, все мы знаем и используем инфиксную нотацию для записи арифметических вычислений, познавая ее с самого раннего детства, зачем же утруждать себя использованием необычной формы записи, когда та, что мы использовали, выглядит вполне надежной? Мы – порождения привычек, и если нам не объяснить, почему те или иные особенности так важны, мы упорно будем предпочитать привычное и устоявшееся.

Обе эти черты внесены в Clojure не как дань памяти предыдущим реализациям Lisp; они дают значительные преимущества, которые стоят небольшой смены привычек.

- Повсеместное использование префиксных операций существенно упрощает синтаксис языка и устраняет неоднозначность в сложных выражениях.
- Использование скобок (для представления списков в текстовой форме) является результатом *гомоиконности* (homoiconic) языка Clojure. Что это означает, подробнее рассказывается в разделе «Гомоиконность», ниже, но последствия этого многообразны: гомоиконность позволяет создавать и использовать предметно-ориентированные языковые конструкции и конструкции для метапрограммирования, что просто невозможно в языках программирования, не обладающих гомоиконностью.

После начального периода привыкания вы, весьма вероятно, обнаружите, что синтаксис языка Clojure требует меньше умственного напряжения при чтении и написании программного кода. Например: сможете ли вы быстро вспомнить какой из операторов в Java, `<<` (поразрядный сдвиг влево) или `&` (поразрядное И), имеет более высокий приоритет? Каждый раз программист вынужден приостановиться, чтобы вспомнить порядок выполнения операторов (или заглянуть в руководство), каждый раз ему приходится отступить на шаг назад и добавить группирующие круглые скобки «на всякий случай». И каждый раз, когда программист забывает подумать об этом, увеличивается вероятность появления ошибки в его коде. Представьте язык, при использовании которого не приходится беспокоиться о порядке выполнения операторов; Clojure – именно такой язык.

Вы можете сказать: «Но в нем так часто используются круглые скобки!». И будете неправы.

Там, где это уместно, Clojure заимствует синтаксис представления литералов данных из других языков, таких как Ruby. В отличие от других диалектов Lisp, возможно знакомых вам и повсеместно использующих списки в круглых скобках, Clojure обладает богатым набором литералов для определения данных и коллекций, таких как векторы, ассоциативные массивы (maps), множества и списки, а также таких конструкций, как записи (близко напоминающие структуры в языке C).

Если подсчитать и сравнить количество ограничительных символов и зарезервированных слов ограничителей всех видов (`()`, `[]`, `{}`, `||` и `end` в Ruby, и так далее) в программах на языках Clojure, Java, Ruby и Python сопоставимого размера, окажется что в программах на языке Clojure их ненамного больше, а нередко и меньше, благодаря его лаконичности.

Выражения, операторы, синтаксис и очередность

Весь программный код на Clojure состоит из выражений, каждое из которых возвращает единственное значение. Этим он отличается от многих других языков, имеющих инструкции, не возвращающие значений, такие как `if`, `for` и `continue`, и используемые для управления потоком выполнения программы. В Clojure все эти языковые конструкции являются *выражениями*, возвращающими некоторое значение.

Выше уже было показано несколько примеров выражений на языке Clojure:

- `60`;
- `[60 80 100 400]`;
- `(average [60 80 100 400])`;
- `(+ 1 2)`.

Все эти выражения возвращают единственное значение. Правила их вычисления чрезвычайно просты, в сравнении с другими языками программирования:

1. Списки (ограничиваются круглыми скобками) являются вызовами. Первое значение в списке интерпретируется как оператор, а остальные – как параметры. Первый элемент списка часто называют *позицией функции* (function position), потому что в нем передается функция или символ, указывающий на функцию для вызова. Выражение вызова получает значение, возвращаемое функцией.
2. Символы (такие как `average` или `+`) преобразуются в именованное значение, находящееся в текущей области видимости. Это значение может быть функцией, локальными данными (как вектор `numbers` в функции `average`), Java-классом, макросом или специальной формой. О макросах и специальных формах мы поговорим чуть ниже, а пока просто считайте их функциями.
3. Все остальные выражения возвращают буквальные значения, которые они описывают.

Примечание. Списки в Lisp часто называют s-выражениями (s-expressions или sexprs), то есть символическими выражениями (symbolic expressions), из-за важной роли, которую играют символы в определении значений для использования в вызовах, выраженных такими списками. В общем случае допустимым s-выражением считается такое выражение, в результате вычисления которого получается то, что называется формой: например, выражение `(if condition then else)` является `if`-формой, выражение `[60 80 100 400]` формой вектора. Однако не все s-выражения являются формами: выражение `(1 2 3)` является допустимым s-выражением – списком из трех целых чисел – но попытка выполнить его закончится ошибкой, потому что первое значение в списке является целым числом, которое нельзя вызвать.

Второй и третий пункты в равной степени можно распространить на большинство других языков (хотя, литералы в языке Clojure более выразительны, как будет показано ниже). Однако рассмотрение

вызовов, написанных на других языках, позволяет быстро убедиться в сложности их синтаксиса.

Таблица 1.1. Сравнение синтаксиса вызовов функций в Clojure, Java, Python и Ruby

Выражение на языке Clojure	Эквивалент на языке Java	Эквивалент на языке Python	Эквивалент на языке Ruby
`(not k)`	`!k`	`not k`	`not k` или `! k`
`(inc a)`	`a++, ++a, a += 1, a + 1`[a]	`a += 1, a + 1`	`a += 1`
`(/ (+ x y) 2)`	`(x + y) / 2`	`(x + y) / 2`	`(x + y) / 2`
`(instance? java.util.List al)`	`al instanceof java.util.List`	`isinstance(al, list)`	`al.is_a? Array`
`(if (not a) (inc b) (dec b))`[b]	`!a ? b + 1 : b - 1`	`b + 1 if not a else b - 1`	`!a ? b + 1 : b - 1`
`(Math/pow 2 10)`[c]	`Math.pow(2, 10)`	`pow(2, 10)`	`2 ** 10`
`(.someMethod someObj "foo" (.otherMethod otherObj 0))`	`someObj.someMethod ("foo", otherObj. otherMethod(0))`	`someObj. someMethod ("foo", otherObj. otherMethod(0))`	`someObj.someMethod ("foo", otherObj. otherMethod(0))`

[a] Операции инкремента и декремента на месте не имеют прямого эквивалента в языке Clojure, потому что в нем не поддерживается прямая модификация значений переменных. Преимущества такой организации подробно описываются в главе 2, в частности в разделе «О важности значений».

[b] Не забывайте, что формы управления потоком выполнения, включая `if` и `when`, в языке Clojure возвращают значения, подобно любым другим выражениям. Здесь значением выражения `if` будет либо `(inc b)`, либо `(dec b)`, в зависимости от значения выражения `(not a)`.

[c] Это первый пример вызова библиотечных методов Java из Clojure. Подробности см. в главе 9.

Обратите внимание на запутанный синтаксис вызовов (здесь мы имеем в виду Java, однако синтаксис языков Python и Ruby мало чем отличается):

- инфиксные операторы присутствуют (например, `a + 1`, `al instanceof List`), но в любом нетривиальном программном коде приходится использовать некоторое, порой значительное, количество круглых скобок, чтобы явно определить порядок выполнения операторов;
- унарные операторы допускают два варианта записи, префиксный (например, `!k` и `++a`) и постфиксный (например, `a++`);
- вызовы статических методов включают префикс, такой как `Math.pow(2, 10)`, но...;

- при вызове методов экземпляров используется необычное многообразие инфиксных позиций: имя метода, объект (чей метод вызывается) и параметры[1].

В языке Clojure, напротив, все выражения следуют одному простому правилу: первое значение в списке является оператором, остальные – параметрами этого оператора. Здесь отсутствуют инфиксные или постфиксные операторы, и отсутствуют сложные для запоминания правила, определяющие порядок выполнения операторов. Поэтому синтаксис языка проще изучать и использовать, а код на таком языке легче читать.

Гомоиконность

Формат записи выполняемого кода совпадает с форматом записи данных. Эта особенность формально называется *гомоиконностью* (homoiconicity), или неформально – *код-это-данные*[2]. Такое существенное упрощение в сравнении с другими языками обеспечивает более широкие возможности метапрограммирования, чем в негомоиконных языках. Чтобы понять причину, необходимо немного поговорить о языках вообще и о том, как их программный код соотносится с внутренним его представлением.

Напомню, что первым шагом, выполняемым интерактивной оболочкой REPL, является *чтение* программного кода, введенного вами. Все языки программирования должны обеспечивать преобразование текстового представления программного кода в некоторое представление, пригодное для компиляции или выполнения. В большинстве языков программирования текст преобразуется в *абстрактное синтаксическое дерево* (Abstract Syntax Tree, AST), или АСТ. Этот термин звучит сложнее, чем есть на самом деле: АСТ – это структура данных, являющаяся формальным представлением исходного про-

---

[1] В языке Python для методов экземпляра используется аналогичная инфиксная форма записи, но она отличается от принятой в Algol-подобных языках тем, что Python требует явно указывать имя первого параметра, обычно `self`.

[2] Clojure – не единственный гомоиконный язык, и само понятие гомоиконности не является новым. К числу других гомоиконных языков относятся другие диалекты языка Lisp, все разновидности машинных языков (и по этой причине – язык Ассемблера), Postscript, XSLT и XQuery, Prolog, R, Factor, Io и другие.

граммного кода. Например, на рис. 1.1 показано несколько примеров преобразования исходных текстов в синтаксические деревья[1].

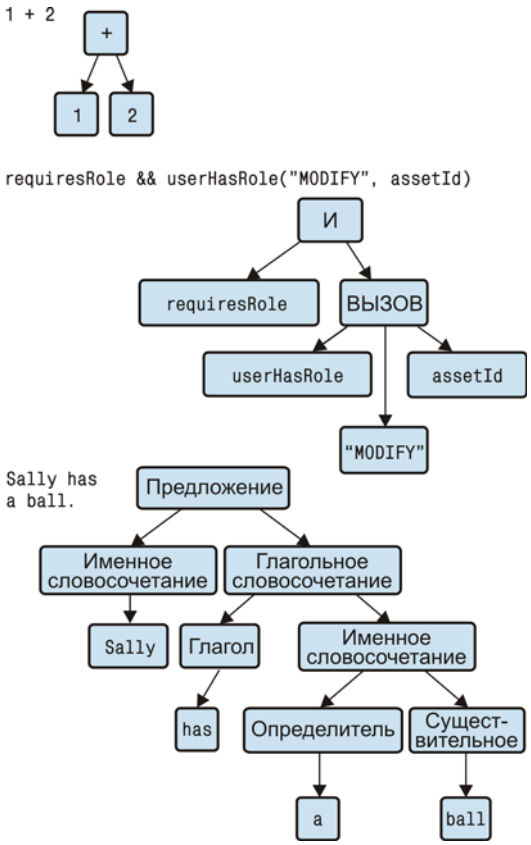

Рис. 1.1. Примеры преобразования исходных текстов в формальные модели

Такие преобразования исходных текстов в синтаксические деревья лежат в основе определения языка, определяют его выразительность и соответствие правилам выбранной предметной области. Именно этим и объясняется привлекательность предметно-ориенти-

---

[1] Синтаксические деревья на рис. 1.1 были взяты почти полностью со страницы в Википедии: http://en.wikipedia.org/wiki/Parse_tree.

рованных языков (Domain-Specific Language, DSL): если язык был создан специально для решения задач в данной предметной области, то специалистам в этой области проще будет выразить решения своих задач, чем на языке общего назначения.

Недостаток такого подхода состоит в том, что большинство языков не предоставляют возможности контролировать процесс создания деревьев АСТ, то есть, соответствие между исходными текстами и синтаксическими деревьями определяется исключительно разработчиками компилятора/интерпретатора языка. Это вынуждает опытных программистов придумывать хитрые обходные решения, чтобы повысить выразительность языка, с которым они работают:

- генерация кода;
- макросы и препроцессоры (уже которое десятилетие успешно используемые в C и C++);
- расширения компилятора (как в Scala, проект Lombok для Java, трансформации деревьев АСТ в Groovy и Template Haskell).

Эти усложнения введены, чтобы компенсировать тот факт, что авторы языков больше внимания уделяют синтаксису, делая формальные модели (если они вообще доступны) зависимыми от конкретной реализации интерпретатора/компилятора.

В языке Clojure (как и в других диалектах Lisp) используется иной подход: вместо определения синтаксиса программного кода, который должен преобразовываться в дерево АСТ, программы пишутся с использованием структур данных языка Clojure, непосредственно представляющих АСТ. Взгляните на выражение `requiresRole...`, представленное на рис. 1.1, версия кода на Clojure буквально опи-

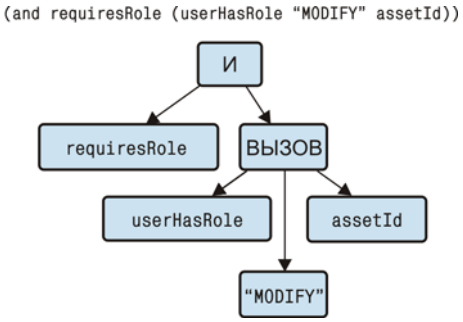

Рис. 1.2. Результат преобразования выражения в дерево АСТ

сывает АСТ этого примера (если помнить о механизме вызовов при обработке списков Clojure).

Тот факт, что *программы на языке Clojure представлены как данные*, означает, что они могут использоваться для создания и преобразования других программ на языке Clojure. Это составляет основу механизма макросов – инструмента метапрограммирования на языке Clojure – совершенно отличающегося от грубого и неуклюжего механизма макросов в стиле языка C и в других препроцессорах исходных текстов, и является идеальным инструментом, когда выразительность и предметная ориентированность имеют первоочередное значение. Детально поддержка макросов в языке Clojure будет рассматриваться в главе 5.

С практической точки зрения прямое соответствие между кодом и данными означает, что программный код на языке Clojure, который вводится в интерактивной оболочке REPL или присутствующий в исходном файле, строго говоря не является текстом: программирование выполняется с применением литералов структур данных языка Clojure. Вспомним простую функцию вычисления среднего арифметического из примера 1.2:

```
(defn average
    [numbers]
    (/ (apply + numbers) (count numbers)))
```

Это не просто фрагмент текста, который волшебным образом, преобразуется в определение функции. Это – структура данных списка, содержащая четыре значения: символ `defn`, символ `average`, вектор, содержащий символ `numbers`, и еще один список, описывающий тело функции. После обработки этого списка будет определена функция `average`.

Механизм чтения

Хотя компиляция и выполнение программ происходит с использованием структур данных языка Clojure, исходный программный код все еще принято хранить в виде текстовых файлов. Поэтому, необходим некоторый способ получения структур данных из исходного текста. Эта задача решается *механизмом чтения* языка Clojure.

Действие механизма чтения полностью определяется единственной функцией `read`, которая читает текстовое содержимое из потока

символов[1] и возвращает структуру данных, закодированную в нем. Именно этот механизм используется интерактивной оболочкой Clojure REPL для чтения ввода. Каждая законченная структура данных затем передается для обработки среде выполнения Clojure.

Для целей исследования удобнее использовать функцию `read-string`, которая действует так же как `read`, но данными для нее служит строковый аргумент:

```
(read-string "42")
;= 42
(read-string "(+ 1 2)")
;= (+ 1 2)
```

Механизм чтения по сути выполняет *десериализацию* (deserialization). Структуры данных и другие литералы имеют в языке Clojure определенные текстовые представления, которые механизм чтения десериализует в соответствующие значения и структуры данных.

Возможно, вы обратили внимание, что значения в REPL выводятся точно так же, как были введены: числа и другие атомарные литералы выводятся в ожидаемом формате, списки заключены в круглые скобки, векторы – в квадратные скобки, и так далее. Это объясняется наличием функций `pr` и `pr-str`, парных функциям `read` и `read-string`, первая из которых выводит значения Clojure в текстовом представлении в `*out*`[2], а вторая возвращает строку с текстовым представлением, которое в свою очередь можно передать функции `read`. Поэтому, структуры данных и значения в языке Clojure легко сериализуются и десериализуются в представления, понятные человеку и механизму чтения:

```
(pr-str [1 2 3])
;= "[1 2 3]"
(read-string "[1 2 3]")
;= [1 2 3]
```

---

[1] Технически, функция `read` использует класс `java.io.PushbackReader`.

[2] По умолчанию значение `*out*` ссылается на `stdout`, но легко может быть перенаправлено в другой поток вывода. Пример перенаправления можно найти в разделе «Создание простой системы журналирования (logging) с функциями высшего порядка», в главе 2.

Примечание. В программах на Clojure механизм чтения часто используется для сериализации, вместо XML, интерфейса `java.io.Serializable`, или других способов передачи программных структур, особенно когда нужен формат, понятный людям.

Скалярные литералы

Скалярные литералы – это синтаксис представления значений, не являющихся коллекциями. Большинство из них являются значениями распространенных типов, имеющихся в Java или аналогичных в Ruby, Python и других языках. Другие характерны для языка Clojure и несут новую семантику.

Строки

Строки в языке Clojure являются обычными Java-строками (то есть, экземплярами класса `java.lang.String`) и в текстовом представлении точно так же заключаются в двойные кавычки:

```
"hello there"
;= "hello there"
```

Строковые литералы в языке Clojure могут занимать несколько строк, без использования особого синтаксиса (как, например, в Python):

```
"multiline strings
are very handy"
;= "multiline strings\nare very handy"
```

Логические значения

Для обозначения логических значений в языке Clojure используются лексемы `true` и `false`, так же как в Java, Ruby и Python (в последнем из них эти лексемы записываются, начиная с символа верхнего регистра).

nil

Значение `nil` в Clojure соответствует значению `null` в Java, `nil` в Ruby и `None` в Python. Кроме того, в условных инструкциях значение `nil` в языке Clojure интерпретируется как `false`, точно так же, как в языках Ruby и Python.

Знаки (characters)

Символьные литералы начинаются с обратного слеша:

```
(class \c)
;= java.lang.Character
```

Символьные литералы могут также записываться в формате Юникода и восьмеричной форме:

```
\u00ff
;= \ÿ
\o41
;= \!
```

Кроме того, существует несколько символьных литералов со специальными именами для обозначения часто используемых символов, которые выводятся как пробельные:

- \space;
- \newline;
- \formfeed;
- \return;
- \backspace;
- \tab.

Ключевые слова (keywords)

Ключевые слова обозначают сами себя и часто используются как средство доступа к значениям в коллекциях и типах, таких как ассоциативные массивы и записи:

```
(def person {:name "Sandra Cruz"
             :city "Portland, ME"})
;= #'user/person
(:city person)
;= "Portland, ME"
```

Здесь создается ассоциативный массив с двумя элементами, :name и :city, и затем выполняется поиск по ключу :city. Такое возможно, потому что ключевые слова – это функции, отыскивающие сами себя в коллекциях, передаваемых в аргументе.

Синтаксически ключевые слова всегда начинаются с двоеточия и могут содержать любые непробельные знаки. Знак слеша (/) исполь-

зуется для обозначения *ключевых слов, принадлежащих пространствам имен*, а ключевые слова, начинающиеся с двух двоеточий (::), интерпретируются механизмом чтения, как принадлежащие текущему пространству имен, или другому, если ключевое слово начинается с псевдонима пространства имен, например, ::alias/kw. Это напоминает механизм использования пространств имен в XML. То есть, одинаковые имена могут использоваться для хранения значений с разной семантикой[1]:

```
(def pizza {:name "Ramunto's"
            :location "Claremont, NH"
            ::location "43.3734,-72.3365"})
;= #'user/pizza
pizza
;= {:name "Ramunto's", :location "Claremont, NH", :user/location "43.3734,-72.3365"}
(:user/location pizza)
;= "43.3734,-72.3365"
```

Это позволяет разным модулям в одном приложении и различным группам в одной организации безопасно использовать одинаковые имена без применения сложных предметных моделей или соглашений, таких как использование подчеркиваний для конфликтующих имен.

Ключевые слова – это разновидность «именованных» значений, потому что они имеют встроенные имена, доступные при помощи функции name, и встроенные пространства имен, доступные при помощи функции namespace:

```
(name :user/location)
;= "location"
(namespace :user/location)
;= "user"
(namespace :location)
;= nil
```

Другой разновидностью именованных значений являются символы (symbols).

---

[1] Ключевые слова, принадлежащие пространствам имен, также часто используются с мультиметодами (multimethods) и иерархиями isa?, о которых подробно рассказывается в главе 7.

Символы (symbols)

Если ключевые слова – это идентификаторы в пределах коллекций (таких как ассоциативные массивы и записи), то *символы* – это идентификаторы, действующие в пределах всей среды выполнения Clojure. В их число входят переменные (vars), которые являются именованными блоками памяти для хранения функций и данных, Java-классы, локальные ссылки и так далее. Вернемся к предыдущему примеру 1.2:

```
(average [60 80 100 400])
;= 160
```

`average` – это символ, ссылка на функцию, хранящаяся в переменной с именем `average`.

Символы должны начинаться с нечислового знака и помимо букв и цифр могут содержать `*`, `+`, `!`, `-`, `_` и `?`. Символы, содержащие слеш (`/`), обозначают *символы с пространством имен* (namespaced symbols) и соответствуют именованным значениям в указанном пространстве имен. Результат интерпретации символов в сущности, на которые они ссылаются, зависит от контекста вычислений и пространств имен, доступных внутри этого контекста. Подробнее о семантике пространств имен и интерпретации символов рассказывается в разделе «Пространства имен» ниже.

Числа

В языке Clojure поддерживается множество числовых литералов (табл. 1.2). Многие из них весьма обычны, но есть и такие, которые редко встречаются в языках программирования общего назначения и применение которых может упростить реализацию отдельных алгоритмов, особенно когда алгоритмы определяются в терминах определенных способов представления чисел (восьмеричные и двоичные числа, рациональные числа и числа в экспоненциальном представлении).

> **Внимание.** Среда выполнения Java определяет набор примитивных типов для чисел и Clojure поддерживает совместимость с ними, однако предпочтение отдается типам `long` и `double` перед числовыми значениями других размеров, включая байты, короткие целые, целые и вещественные одинарной точности. Это означает, что для взаимодействий с другими программными компонентами (например, в вызовах Java-методов), из литералов или данных среды выполнения при необходимости будут создаваться значения примитивных типов меньшего размера, а в остальных случаях будут использоваться типы `long` и `double`.

Чаще об этом не приходится беспокоиться. Но, если нужна высокая точность вычислений, внимательно прочитайте главу 11, где подробно обсуждаются приемы повышения точности операций с примитивными типами в языке Clojure и другие темы, связанные с математическими вычислениями.

Таблица 1.2. Числовые литералы в языке Clojure

Литерал	Числовой тип
`42`, `0xff`, `2r111`, `040`	длинное целое (64-разрядное целое со знаком)
`3.14`, `6.0221415e23`	вещественное двойной точности (64-разрядное вещественное с плавающей точкой в формате IEEE)
`42N`	`clojure.lang.BigInt` (целое число произвольной точности[a])
`0.01M`	`java.math.BigDecimal` (вещественное число произвольной точности)
`22/7`	`clojure.lang.Ratio`

[a] значение типа `clojure.lang.BigInt` автоматически преобразуется в значение типа `java.math.BigInteger` при необходимости. Подробнее об интерпретации числовых литералов в языке Clojure рассказывается в главе 11.

Знак любого числового литерала можно изменить на обратный, добавив перед ним дефис (-).

Ниже перечислены некоторые наиболее интересные числовые литералы.

Шестнадцатеричные числа

Как и большинство языков программирования, Clojure поддерживает шестнадцатеричную форму записи целочисленных значений; `0xff` обозначает число `255`, `0xd055` – число `53333`, и так далее.

Восьмеричные числа

Числовые литералы, начинающиеся с нуля, интерпретируются как числа в восьмеричной форме записи. Например, восьмеричное число `040` – это десятичное число `32`.

Числа в произвольной системе счисления

В языке Clojure поддерживается возможность определять целочисленные литералы в произвольной системе счисления, используя формат записи *`BrN`*, где *`N`* – цифры, представляющие число, а *`B`* – основание системы счисления, в которой должно интерпретироваться число *`N`*. То есть, для записи двоичных

чисел можно использовать префикс `2r` (`2r111` – это число `7`), `16r` – для записи шестнадцатеричных чисел (`16rff` – это число `255`), и так далее. Поддерживаются системы счисления с основаниями, вплоть до `36`[1].

Числа с произвольной точностью представления
Любые числовые литералы (кроме рациональных чисел) могут определяться с произвольной точностью, добавлением соответствующего окончания: для десятичных чисел – символ `M`, для целых чисел – символ `N`. Полное описание вы найдете в разделе «Ограниченная и произвольная точность», в главе 11.

Рациональные числа
В языке Clojure имеется непосредственная поддержка рациональных чисел и их литералов. Литералы рациональных чисел всегда состоят из двух целых чисел, разделенных слешем (/).

Полное обсуждение поддержки рациональных чисел и их взаимодействие с другими числовыми типами языка Clojure можно найти в разделе «Рациональные числа», в главе 11.

Регулярные выражения

Строки, начинающиеся со знака «диез» (#), механизм чтения языка Clojure интерпретирует как литералы регулярных выражений:

```
(class #"(p|h)ail")
;= java.util.regex.Pattern
```

Синтаксис литералов регулярных выражений в Clojure в точности повторяет синтаксис `/.../` регулярных выражений в Ruby, за исключением несущественных различий в символах-ограничителях. Фактически, реализации поддержки регулярных выражений в Ruby и Clojure очень близки:

---

[1] Предел реализации `java.math.BigInteger`. Обратите внимание: несмотря на то, что для парсинга литералов используется тип `BigInteger`, конкретный тип числа, возвращаемого механизмом чтения, совместим с другими целочисленными литерами Clojure – либо `long`, либо `BigInt`, если число должно быть представлено с произвольной точностью.

```
# Ruby
>> "foo bar".match(/(...) (...)/).to_a
["foo bar", "foo", "bar"]

;; Clojure
(re-seq #"(...) (...)" "foo bar")
;= (["foo bar" "foo" "bar"])
```

Синтаксис регулярных выражений в Clojure не требует экранирования обратных слешей как в Java:

```
(re-seq #"(\d+)-(\d+)" "1-3") ;; превратится в "(\\d+)-(\\d+)" в Java
;= (["1-3" "1" "3"])
```

Экземпляры `java.util.regex.Pattern`, возвращаемые в результате интерпретации литералов регулярных выражений в Clojure, полностью эквивалентны тем, что создаются в Java, то есть используют в целом превосходную реализацию регулярных выражений `java.util.regex`[1]. Благодаря встроенному в Clojure механизму обращения (interoperability) к Java-платформе, объекты класса `Pattern` можно использовать непосредственно, однако многие найдут более простыми и удобными вспомогательные функции языка Clojure (такие как `re-seq`, `re-find`, `re-matches` и другие, имеющиеся в пространстве имен `clojure.string`).

Комментарии

Механизм чтения различает два типа комментариев.

- Однострочные комментарии начинаются с точки с запятой (`;`). Все, что следует за точкой с запятой до конца строки, игнорируется механизмом чтения. Комментарии этого типа эквивалентны комментариям в Java и JavaScript, начинающимся с пары символов //, и комментариям в Ruby и Python, начинающимся с символа `#`.
- Комментарии *уровня формы* (form-level) определяются с помощью макроса `#_`. Он вынуждает механизм чтения игнорировать форму, следующую за макросом:

[1] Полное описание форм регулярных выражений, поддерживаемых в Java, можно найти в документации javadoc к `java.util.regex.Pattern`, по адресу: http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html.

```
(read-string "(+ 1 2 #_(* 2 2) 8)")
;= (+ 1 2 8)
```

Список из четырех чисел – `(+ 1 2 4 8)` – превращается в список из трех чисел, потому что целая форма умножения игнорируется из-за префикса `#_`.

Поскольку программный код на языке Clojure определяется с использованием литералов структур данных, эта разновидность комментариев в некоторых случаях может оказаться более полезной, чем обычные текстовые комментарии, воздействующие на строки или знаки в определенных позициях (как, например, многострочные комментарии `/* */` в Java и JavaScript). Например, взгляните на применение следующего приема отладки через вывод промежуточной информации в `stdout`:

```
(defn some-function
     [...аргументы...]
     ...код...
     (if ...условие-действия-режима-отладки...
       (println ...отладочная-информация...)
       (println ...еще-отладочная-информация...))
     ...код...)
```

Чтобы исключить из программы эти формы `println`, достаточно просто добавить макрос `#_` механизма чтения перед формой `if` и перезагрузить определение функции. При этом совершенно не имеет значения, сколько строк занимает форма, одну или несколько сотен.

Примечание. В Clojure существует еще один способ комментировать программный код – макрос `comment`:

```
(when true
     (comment (println "hello")))
;= nil
```

Форма `comment` может включать любой объем игнорируемого программного кода, но результат ее интерпретации не игнорируется механизмом чтения, подобно тому, как игнорируется воздействие макроса `#_` на следующую за ним форму. Форма `comment` всегда интерпретируется как значение `nil`. Обычно это не вызывает проблем, но иногда такое поведение оказывается неприемлемым. Взгляните на переделанный пример с макросом `#_`:

```
(+ 1 2 (comment (* 2 2)) 8)
;= #<NullPointerException java.lang.NullPointerException>
```

Проблема обусловлена тем, что макрос `comment` возвращает значение `nil`, которое не является допустимым аргументом для `+`.

Пробелы и запятые

Вы могли обратить внимание на отсутствие запятых между формами, параметрами в вызовах функций, элементами в литералах структур данных, и так далее:

```
(defn silly-adder
    [x y]
    (+ x y))
```

Это объясняется тем, что пробельных символов вполне достаточно для разделения форм и значений, передаваемых механизму чтения. Более того, *запятые интерпретируются механизмом чтения как пробельные знаки*. Например, следующий фрагмент кода эквивалентен фрагменту выше:

```
(defn silly-adder
    [x, y]
    (+, x, y))
```

Чтобы подтвердить правильность вышесказанного:

```
(= [1 2 3] [1, 2, 3])
;= true
```

Использовать или не использовать запятые, это вопрос стиля и личных предпочтений. С другой стороны, их обычно используют, только когда это повышает удобочитаемость программного кода, например, для отделения пар значений, когда в одной строке перечисляется несколько пар[1]:

```
(create-user {:name new-username, :email email})
```

Литералы коллекций

Механизм чтения предусматривает синтаксис определения наиболее используемых структур данных в программах на языке Clojure:

---

[1] Вопросы стиля являются наиболее сложными и неоднозначными, но как бы то ни было, вам редко придется видеть код, где в одной строке определяется две или три пары значений, несколько именованных аргументов (keyword arguments), и так далее.

```
'(a b :name 12.5)       ;; список
['a 'b :name 12.5]      ;; вектор
{:name "Chas" :age 31}  ;; ассоциативный массив
#{1 2 3}                ;; множество
```

Поскольку списки в Clojure используются также для вызовов функций, чтобы предотвратить интерпретацию литералов списков как вызовов функций, их необходимо предварять апострофом (').

Особенности этих структур данных исследуются в главе 3.

Прочий синтаксический сахар механизма чтения

Механизм чтения предоставляет еще несколько синтаксических конструкций, повышающих лаконичность и выразительность программного кода на Clojure.

- Вычисление формы может быть подавлено добавлением перед ней символа апострофа ('). Подробности см. ниже, в разделе «Подавление вычислений: quote».
- Анонимные литералы функций могут определяться более лаконично, с применением формы записи `#()`. Подробности см. ниже, в разделе «Литералы функций».
- Символы во время выполнения замещаются значениями, хранящимися в соответствующих им переменных, однако существует возможность ссылаться на сами переменные, добавляя префикс `#'` перед символами. Подробности см. ниже, в разделе «Ссылки на переменные: var».
- Экземпляры ссылочных типов могут разыменовываться (то есть, возвращать значение, хранящееся в объекте, на который указывает ссылка) за счет добавления префикса `@` к символу имени экземпляра. Подробности см. в разделе «Ссылочные типы в Clojure», в главе 4.
- Механизм чтения предоставляет три специальные синтаксические конструкции для определения макросов: `` ` ``, `~` и `~@`. Подробное исследование макросов можно найти в главе 5.
- С технической точки зрения, поддерживается всего две формы взаимодействий с программным кодом Java, однако механизм чтения предоставляет еще формы, которые в действительности преобразуются в первые две. Подробности см. ниже, в разделе «Взаимодействие с Java: . и new».

- Все структуры данных и ссылочные типы в языке Clojure поддерживают *метаданные* – небольшие фрагменты информации, которые могут быть связаны со значениями или ссылками и не оказывающие влияния на результат таких операций, как сравнение. Приложения могут использовать эти метаданные в самых разных целях, метаданные используются самим языком Clojure там, где другие языки используют ключевые слова (например, чтобы показать, что функция является частной (private) для данного пространства имен, или чтобы указать тип аргумента или возвращаемого значения). Механизм чтения позволяет присоединять метаданные к литеральным значениям с помощью нотации ^. Подробности см. в разделе «Метаданные», в главе 3.

Пространства имен

К настоящему моменту у вас уже должно сложиться представление о том, как действуют наиболее сложные части интерактивной оболочки Clojure REPL (и самого языка Clojure).

- Чтение (read): механизм чтения Clojure читает программный код в текстовом представлении, создает структуры данных (такие как списки, векторы, и так далее) и атомарные значения (такие как символы (symbols), числа, строки и так далее).
- Вычисление (evaluate): многие значения, возвращаемые механизмом чтения, равны сами себе (включая большинство структур данных и скаляров, таких как строки и ключевые слова). Порядок интерпретации списков, как вызовов операторов в *позициях функций*, описывается в разделе «Выражения, операторы, синтаксис и очередность», выше.

Единственное, что осталось, – это разобраться с интерпретацией символов. До сих пор они использовались для именования функций и ссылок на них, локальных значений и так далее. За пределами области действия локальных имен, семантика интерпретации символов тесно связана с пространствами имен, фундаментальной единицей структуризации программного кода в языке Clojure.

Весь программный код Clojure определяется и вычисляется внутри пространства имен. Пространства имен в Clojure являются примерными аналогами модулей в Ruby и Python, или пакетами в Ja-

va[1]. Фактически они связывают между собой *символы* и *переменные* или импортированные Java-классы.

Одним из *ссылочных типов*[2] являются переменные (vars), которые в Clojure представляют хранилища значений любых *ссылочных* типов (`ref`, `atom`, `agent`). Они связаны с символами внутри пространства имен, которые другой код может использовать для их поиска и, соответственно, получения их значений.

Переменные определяются в языке Clojure с помощью специальной формы `def`, действие которой распространяется только на текущее пространство имен[3]. Теперь определим переменную `x` в пространстве имен `user`. Именем этого значения является символ, служащий ключом в текущем пространстве имен:

```
(def x 1)
;= #'user/x
```

Обратиться к значению можно по этому символу:

```
x
;= 1
```

Здесь используется *неквалифицированный* символ `x`, поэтому его поиск выполняется в текущем пространстве имен. Переменные можно переопределять, что очень важно для поддержки разработки в интерактивном режиме в REPL:

```
(def x "hello")
;= #'user/x
x
;= "hello"
```

---

[1] Фактически, пространства имен в точности соответствуют пакетам в Java, когда типы, объявленные в программном коде на Clojure, компилируются в классы Java. Например, тип `Person`, объявленный в пространстве имен `app.entities`, будет преобразован в Java-класс с именем `app.entities.Person`. Более подробную информацию об определении типов и записей в Clojure можно найти в главе 6.

[2] То есть объектов таких типов, как ссылки (ref), атомы (atom) и агенты (agent). Подробное описание ссылочных типов, каждый из которых привносит собственные особенности в механизм параллельного выполнения, приводится в разделе «Ссылочные типы в Clojure», в главе 4.

[3] Не забывайте, что сеанс REPL всегда начинается в пространстве имен по умолчанию `user`.

Внимание. Переменные в языке Clojure не являются переменными в классическом понимании. Переменные должны определяться в интерактивном контексте, таком как REPL, или в исходном файле Clojure, подобно тому, как определяются функции, постоянные значения и прочее. В частности, переменные верхнего уровня (то есть, переменные, глобально доступные в пространствах имен и создаваемые посредством объявления `def` или его вариантами) могут определяться только в выражениях верхнего уровня и никогда внутри функций, вызываемых в ходе нормального выполнения программы.

Дополнительные пояснения см. в разделе «Переменные в языке Clojure не являются переменными в классическом понимании», в главе 4.

Символ может быть указан с использованием пространства имен. В таком случае его поиск будет производиться не в текущем, а в указанном пространстве имен:

```
*ns*                          ❶
;= #<Namespace user>
(ns foo)
;= nil
*ns*
;= #<Namespace foo>
user/x
;= "hello"
x
;= #<CompilerException java.lang.RuntimeException:
;=Unable to resolve symbol: x in this context, compiling:(NO_SOURCE_PATH:0)>
```

❶ Символ `*ns*` всегда ссылается на текущее пространство имен.

В примере выше создается новое пространство имен, с помощью макроса `ns` (который переключает интерактивную оболочку REPL на новое пространство имен), и затем выполняется обращение к значению `x` в пространстве имен `user`. Так как новое пространство имен `foo` было создано только что, в нем отсутствует символ `x`, поэтому попытка обратиться к нему завершается неудачей.

Примечание. Для эффективного использования языка Clojure, необходимо знать, как создавать, определять, организовывать и управлять пространствами имен. Для этих целей существует множество функций, описание которых приводится в разделе «Определение и использование пространств имен» в главе 8.

Выше уже упоминалось, что пространства имен служат для организации связи между символами и импортируемыми Java-классами. Все классы в пакете `java.lang` по умолчанию импортируются во все пространства имен Clojure и потому доступны без указания пакета; для обращения к неимпортированным классам, их имена должны предваряться именами пакетов. Любой символ, являющийся именем класса, преобразуется в этот класс:

```
String
;= java.lang.String
Integer
;= java.lang.Integer
java.util.List
;= java.util.List
java.net.Socket
;= java.net.Socket
```

Кроме того, пространства имен автоматически получают псевдонимы на все переменные, объявленные в основном пространстве имен стандартной библиотеки Clojure, `clojure.core`. Например, функцию `clojure.core.filter` можно вызвать из любого пространства имен, используя только ее имя `filter`:

```
filter
;= #<core$filter clojure.core$filter@7444f787>
```

Это лишь самые основные принципы работы пространств имен в Clojure; подробнее о них и их использовании для структурирования своих проектов можно узнать в разделе «Определение и использование пространств имен» в главе 8.

Интерпретация символов

Получив базовые представления о пространствах имен, можно вновь вернуться к функции `average` из примера 1.2 и выяснить, как точно она интерпретируется:

```
(defn average
    [numbers]
    (/ (apply + numbers) (count numbers)))
```

Как мы узнали в разделе «Гомоиконность», это всего лишь каноническое текстовое представление структуры данных на языке Clojure, содержащей другие данные. В теле этой функции присутствует несколько символов, каждый из которых ссылается либо на переменную в текущем пространстве имен, либо на значение в локальной области видимости:

- `/`, `apply`, `+` и `count` интерпретируются как функции в пространстве имен `clojure.core`;
- имя `numbers` определяет единственный аргумент функции (в виде вектора `[numbers]`)[1], и используется для ссылки на значение этого аргумента в теле функции (при использовании в выражениях `(apply + numbers)` и `(count numbers)`).

Если вспомнить, что списки интерпретируются как вызовы функций, где функции указываются первым параметром, то у вас будет почти полное представление, о том как вычисляются вызовы этих функций:

```
(average [60 80 100 400])
;= 160
```

Символ `average` ссылается здесь на значение `#'average`, переменную в текущем пространстве имен, хранящую определенную нами функцию. Эта функция вызывается с вектором чисел, который в теле функции `average` будет называться `numbers`. В результате операций, выполняемых в теле функции, получается значение `160`, которое возвращается вызывающей программе: в данном случае интерактивной оболочке REPL, которая выводит его в `stdout`.

Специальные формы

Оставив пока в стороне особенности взаимодействия с Java, можно сказать, что символы в позиции функции допускают только две интерпретации:

- Значение именованной переменной или локальной привязки, как было показано выше.

---

[1] Подробнее о том, как определяются функции и их аргументы рассказывается в разделе «Создание функций: fn», ниже.

- *Специальная форма* Clojure[1].

Специальные формы – это базовые строительные блоки, из которых строятся все остальные элементы языка Clojure. Эта особенность унаследована от ранних диалектов Lisp, в которых также определялся ограниченный набор примитивов фундаментальных операций, достаточный для описания любых возможных вычислений[2]. Кроме того, специальные формы имеют собственный синтаксис (например, многие из них не принимают аргументы непосредственно) и механизм вычисления.

Как вы могли видеть выше, многое из того, что в других языках программирования часто описывается как элементарные операции и выражения – включая формы управления, такие как `when`, и операторы, такие как сложение и отрицание – в языке Clojure таковыми не являются. Все, что не является специальной формой, реализуется на языке Clojure с использованием ограниченного набора элементарных операций[3]. С практической точки зрения это означает, что любые конструкции, отсутствующие в Clojure, можно реализовать самостоятельно[4].

Все в языке Clojure построено на основе этих специальных форм и вы должны знать, как действует каждая из них, так как многие из них вам придется использовать постоянно. Поэтому рассмотрим все их по очереди.

---

[1] Специальные формы всегда имеют более высокий приоритет при интерпретации в позиции функции. Например, в программе может быть именованная переменная и локальная привязка, но вы не сможете сослаться на ее значение в позиции функции, хотя в любой другой позиции это вполне возможно.

[2] В статье Пола Грэма (Paul Graham), «The Roots of Lisp» (http://www.paulgraham.com/rootsoflisp.html), приводится краткое, но понятное описание фундаментальных вычислительных операций, выделенных и перечисленных Джоном Маккарти (John McCarthy). Несмотря на то, что этой формализации уже более 50 лет, она с успехом применяется в современном языке Clojure.

[3] Если открыть файл *core.clj* из репозитория с исходными текстами Clojure, можно увидеть, как это реализовано в действительности: все, от `when` и `or`, до `defn` и `=` определено на самом языке Clojure. Фактически, при наличии желания, на основе специальных форм Clojure можно реализовать собственную версию Clojure (или любого другого языка программирования) с нуля.

[4] Обычно подобное расширение синтаксиса требует применения макросов, которые подробно обсуждаются в главе 5.

Подавление вычислений: quote

Форма `quote` подавляет вычисление выражения. Самый доступный пример – символы, если они ссылаются на переменные, то интерпретируются в соответствующие значения. Применение `quote` подавляет этот процесс, поэтому символы преобразуются в самих себя (так же как строки, числа, и так далее):

```
(quote x)
;= x
(symbol? (quote x))
;= true
```

Механизм чтения предоставляет отдельный синтаксис для `quote` – символ апострофа, предшествующий любой форме, действует подобно `quote`:

```
'x
;= x
```

Подавлено может быть вычисление любой формы в языке Clojure, включая структуры данных. В этом случае возвращается структура данных, в которой рекурсивно подавляется вычисление всех ее элементов:

```
'(+ x x)
;= (+ x x)
(list? '(+ x x))
;= true
```

Списки обычно интерпретируются как вызовы функций, однако добавление `quote` перед списком подавляет его интерпретацию, и возвращает сам список, в данном случае – список из трех символов: `'+`, `'x` и `'x`. Обратите внимание, что это в точности то, что получается при конструировании списка «вручную», без использования литерала списка:

```
(list '+ 'x 'x)
;= (+ x x)
```

Примечание. Увидеть, что производит механизм чтения, можно, добавив апостроф перед формой. Углубимся в механизм чтения и посмотрим, что получится, если применить апостроф к форме с апострофом:

```
``x
;= (quote x)
```

Тот же прием можно применить к другим синтаксическим конструкциям механизма чтения:

```
`@x
;= (clojure.core/deref x)
`#(+ % %)
;= (fn* [p1__3162792#] (+ p1__3162792# p1__3162792#))
``(a b ~c)
;= (seq (concat (list (quote user/a))
;=               (list (quote user/b))
;=               (list c)))                    ❶
```

❶ здесь префикс clojure.core пространства имен опущен для большей удобочитаемости.

Блоки кода: do

Конструкция `do` вычисляет все выражения, переданные ей, и в качестве собственного значения возвращает значение последнего выражения. Например:

```
(do
     (println "hi")
     (apply * [4 5 6]))
; hi
;= 120
```

Значения всех выражений, кроме последнего, просто отбрасываются, однако побочные эффекты этих выражений (такие как вывод в стандартный поток вывода, как в данном примере, или изменение состояния объекта, доступного в текущей области видимости) продолжают действовать.

Обратите внимание, что многие другие формы (включая `fn`, `let`, `loop` и `try` – и их производные, такие как `defn`) неявно обертывают свое содержимое конструкцией `do`, чтобы обеспечить возможность вычисления нескольких внутренних выражений. Как, например, в следующей форме `let`, где определяются два локальных значения. Как, например, в следующей форме `let`, где определяются два локальных значения:

```
(let [a (inc (rand-int 6))
      b (inc (rand-int 6))]
  (println (format "You rolled a %s and a %s" a b))
  (+ a b))
```

Эта особенность позволяет вычислять любое количество выражений в контексте формы `let`, последнее из которых определяет значение всей конструкции. Если бы конструкция `let` не обертывала свое тело формой `do`, ее необходимо было бы использовать явно[1]:

```
(let [a (inc (rand-int 6))
      b (inc (rand-int 6))]
  (do
    (println (format "You rolled a %s and a %s" a b))
    (+ a b)))
```

Определение переменных: def

Мы уже видели форму `def` в действии[2]. Она определяет (или переопределяет) переменную в текущем пространстве имен:

```
(def p "foo")
;= #'user/p
p
;= "foo"
```

Многие другие формы неявно создают или переопределяют переменные и, как следствие, используют форму `def`. Это характерно для форм, начинающихся с префикса «def», таких как `defn`, `defn-`, `defprotocol`, `defonce`, `defmacro`, и так далее.

> **Внимание.** Формы, создающие или переопределяющие переменные, начинаются с префикса «def», но не все формы, начинающиеся с «def» создают или переопределяют переменные. В числе последних можно назвать `deftype`, `defrecord` и `defmethod`.

---

[1] Альтернативным решением для `let` (и всех других форм, неявно использующих `do`) была бы своя реализация семантики «вычисления нескольких выражений и возврата значения последнего выражения», что едва ли можно признать разумным.

[2] См. раздел «Пространства имен» выше, где обсуждается типичное использование переменных в качестве ссылок на значения в пространствах имен. См. также раздел «Переменные» в главе 4, где демонстрируются разные варианты их использования, включая самые сложные, связанные с динамическими областями видимости и ссылками на значения, локальные для потоков выполнения.

Связывание локальных значений: let

Форма `let` позволяет определять именованные ссылки, доступные только в этой форме. Например, следующий простейший статический метод на Java:

```
public static double hypot (double x, double y) {
     final double x2 = x * x;
     final double y2 = y * y;
     return Math.sqrt(x2 + y2);
}
```

эквивалентен следующей функции на Clojure:

```
(defn hypot
  [x y]
  (let [x2 (* x x)
        y2 (* y y)]
    (Math/sqrt (+ x2 y2))))
```

Здесь `x2` и `y2` являются локальными символами, доступными только в теле функции/метода и служат цели создания именованных ссылок с ограниченной областью видимости на промежуточные значения.

> **Примечание.** Для обозначения именованных ссылок, создаваемых формой `let` в языке Clojure, используют разные термины:
> - локальные символы;
> - локальные привязки (local bindings);
> - приватные значения, которые называются let-связками (let-bound).
>
> Связывание в форме `let` полностью отличается от действия макроса `binding`, который управляет областью видимости переменных, локальных для потоков выполнения; подробнее о последнем рассказывается в разделе «Динамическая область видимости» главы 4.

Обратите внимание, что форма `let` неявно используется везде, где требуются локальные символы. В частности `fn` (и, соответственно, все другие формы создания и определения функций, такие как `defn`) использует `let` для связывания параметров функции с локальными символами в области видимости определяемой функции. Например, символы `x` и `y` в функции `hypot` выше, являются let-связками (let-bound), создаваемыми формой `defn`. То есть, вектор, определяющий множество привязок для области видимости `let`, следует той

же семантике, как для определения параметров функции, так и для определения вспомогательных локальных символов.

Примечание. Иногда бывает необходимо вычислить выражение, присутствующее в векторе привязки, который передается форме `let`, но его результат можно игнорировать. В таких случаях принято связывать ненужное значение с именем, состоящим из единственного знака подчеркивания, чтобы механизм чтения программного кода знал, что результат выражения игнорируется преднамеренно.
Использовать такой прием имеет смысл, только когда выражение имеет побочный эффект, например, выводит некоторое промежуточное значение:

```
(let [location (get-lat-long)
      _ (println "Current location:" location)
      location (find-city-name location)]
  ...вывести название города для текущего местоположения...)
```

Здесь извлекаются текущая широта и долгота, с использованием гипотетического API, которые выводятся перед их преобразованием в название города. Мы можем неоднократно присваивать новые значения одному имени в векторе привязок формы `let`, для сохранения промежуточных значений. Чтобы вывести промежуточное значение, мы добавили операцию вывода в вектор привязки перед повторным связыванием имени, но при этом мы указали, что преднамеренно игнорируем значение, возвращаемое выражением, связав это выражение с именем `_`.

Семантика `let` имеет две особенности, существенно отличающие локальные символы в Clojure от локальных переменных в других языках:

1. Все локальные значения *неизменяемы*. Локальный символ можно *переопределить* через использование вложенной формы `let` или повторное присваивание в том же векторе привязок. После обработки вектора привязок нельзя будет изменить локальное значение в пределах одной формы `let`. Это устраняет источник глупых ошибок без ограничения возможностей программиста:
   - для циклов, когда значения должны изменяться в каждой итерации, предоставляются специальные формы `loop` и `recur`; см. раздел «Организация циклов: loop и recur» ниже;
   - для случаев, когда действительно необходимы «изменяемые» локальные значения, Clojure предоставляет множество ссылочных типов; см. раздел «Ссылочные типы в языке Clojure» в главе 4.
2. Вектор привязок формы `let` интерпретируется на этапе компиляции с целью обеспечить возможность *деструктуризации*

(destructuring) наиболее распространенных типов коллекций. Деструктуризация может оказать существенную помощь в устранении излишнего (и порой бессмысленного) кода при работе с коллекциями, переданными в виде аргументов функций.

Деструктуризация (let, часть 2)

Программирование на языке Clojure в значительной степени связано с обработкой различных реализаций абстрактных структур данных, основными примерами которых могут служить *упорядоченные коллекции* (sequential collections) и *ассоциативные массивы* (maps). Многие функции в языке Clojure принимают и возвращают абстрактные коллекции, а не какие-то конкретные их реализации, и большинство библиотек Clojure и приложений созданы с опорой на эти абстракции, а не на конкретные структуры, классы, и так далее. Это помогает конструировать функции и библиотеки с минимумом служебного, «связующего» кода для работы с данными.

Одна из сложностей при работе с абстрактными коллекциями заключается в том, чтобы обеспечить лаконичность обращения к множественным значениям в этих коллекциях. Например, следующая коллекция является вектором Clojure:

```
(def v [42 "foo" 99.2 [5 12]])
;= #'user/v
```

Рассмотрим несколько способов доступа к значениям в этом векторе:

```
(first v)       ❶
;= 42
(second v)
;= "foo"
(last v)
;= [5 12]
(nth v 2)       ❷
;= 99.2
(v 2)           ❸
;= 99.2
(.get v 2)      ❹
;= 99.2
```

❶ Clojure предоставляет вспомогательные функции для доступа к первому (`first`), второму (`second`) и последнему (`last`) значениям в упорядоченной коллекции.
❷ Функция `nth` позволяет получить любое значение из упорядоченной коллекции, используя индекс в этой коллекции.
❸ Векторы являются функциями, принимающими индекс в виде аргумента.
❹ Все упорядоченные коллекции в языке Clojure реализуют интерфейс `java.util.List`, поэтому для доступа к их содержимому можно использовать метод `.get` интерфейса.

Все эти механизмы прекрасно подходят для получения доступа к единственному «верхнеуровнему» значению в векторе, но дело осложняется, когда при выполнении некоторых операций требуется получить доступ сразу к нескольким значениям:

```
(+ (first v) (v 2))
;= 141.2
```

или к значениям во вложенных коллекциях:

```
(+ (first v) (first (last v)))
;= 47
```

Механизм деструктуризации Clojure обеспечивает лаконичный синтаксис декларативного разложения коллекций на составные части и связывания значений, содержащихся в них, с локальными именами в форме `let`. А так как возможность деструктуризации является особенностью `let`, ее можно использовать в любых выражениях, неявно применяющих форму `let` (в таких как `fn`, `defn`, `loop`, и так далее).

Существует две разновидности деструктуризации: одна применяется к упорядоченным коллекциям, а другая – к ассоциативным массивам.

Деструктуризация упорядоченных коллекций

Деструктуризация упорядоченных коллекций (sequential destructuring) применяется к любым упорядоченным коллекциям, включая:

- списки, векторы и последовательности Clojure;
- любые коллекции, реализующие интерфейс `java.util.List` (подобно `ArrayLists` и `LinkedLists`);
- массивы Java;
- строковые значения, которые разлагаются на знаки.

Ниже приводится простой пример, где выполняется деструктуризация значения v, приведенного выше:

Пример 1.3. Простейшая деструктуризация упорядоченной коллекции

```
(def v [42 "foo" 99.2 [5 12]])
;= #'user/v
(let [[x y z] v]
     (+ x z))
;= 141.2
```

В простейшем случае вектор, передаваемый форме let, содержит пары имен и значений, но здесь вместо скалярного имени символа мы передаем вектор символов ([x y z]). Это вызывает деструктуризацию значения v в теле формы let и связывание первого значения с символом x, второго значения – с y, и третьего – с z. Затем эти локальные значения, полученные в результате деструктуризации, можно использовать как любые другие локальные значения. Следующая форма связывания эквивалентна предыдущей:

```
(let [x (nth v 0)
      y (nth v 1)
      z (nth v 2)]
  (+ x z))
;= 141.2
```

Примечание. В языке Python имеется механизм «распаковки», напоминающий механизм деструктуризации упорядоченных последовательностей в языке Clojure. Эквивалентный фрагмент программного кода на Python мог бы выглядеть примерно так:

```
>>> v = [42, "foo", 99.2, [5, 12]]
>>> x, y, z, a = v
>>> x + z
141.19999999999999
```

Аналогичная возможность имеется в языке Ruby:

```
>> x, y, z, a = [42, "foo", 99.2, [5, 12]]
[42, "foo", 99.2, [5, 12]]
>> x + z
141.2
```

Clojure, Python и Ruby имеют некоторые внешние сходства, но, как будет показано далее, Clojure идет намного дальше.

Формы деструктуризации отражают структуру связываемых коллекций[1]. То есть, применяя форму деструктуризации к коллекции можно получить точное представление о том, как значения будут связаны с именами[2]:

```
[x y z]
[42 "foo" 99.2 [5 12]]
```

Формы деструктуризации могут быть составными, благодаря чему легко можно извлечь значения из вектора, вложенного в v[3]:

```
(let [[x _ _ [y z]] v]
(+ x y z))
;= 59
```

Если визуально расположить форму деструктуризации в одну линию над вектором, результат ее действия станет более очевидным:

```
[x  _     _    [y z ]]
[42 "foo" 99.2 [5 12]]
```

> **Внимание.** Вектор, вложенный в исходный вектор, также можно деструктурировать. Механизм деструктуризации не имеет ограничений на глубину вложенности структур данных, но есть пределы разумному применению. Если использовать деструктуризацию для извлечения значений из коллекции на четвертом или более уровне вложенности, форма деструктуризации может оказаться слишком сложной для того, кто будет разбираться с вашим программным кодом, даже если это будете вы сами!

Механизм деструктуризации упорядоченных коллекций имеет две дополнительные особенности, о которых вы должны знать:

---

[1] Термин *деструктуризация* (destructuring) является *обратным* (*de-*) к термину, обозначающему *создание структуры* (structure) данных.

[2] Значения в исходной коллекции, для которых не указаны имена, просто не привязываются в контексте формы `let`; от вас не требуется полностью повторять структуру исходной коллекции, но начало коллекции должно быть «привязано» обязательно.

[3] Еще раз обратите внимание, что подчеркивания (_) в данной форме деструктуризации соответствуют игнорируемым значениям, по аналогии с идиомой, обсуждавшейся выше в этой главе.

Извлечение оставшихся значений из упорядоченных последовательностей

Чтобы извлечь все оставшиеся значения из упорядоченной коллекции, находящиеся за последней именованной позицией в форме деструктуризации, можно воспользоваться оператором `&`. По своему действию он напоминает механизм, лежащий в основе списков аргументов переменной длины (varargs) в методах Java, и является основой для извлечения оставшихся аргументов в функциях на языке Clojure:

```
(let [[x & rest] v]
rest)
;= ("foo" 99.2 [5 12])
```

Это особенно удобно, в частности, для обработки элементов последовательности с использованием рекурсивных вызовов функций или в сочетании с формой `loop`. Обратите внимание, что значением символа `rest` здесь является последовательность, а *не* вектор, даже при том, что форме деструктуризации передается вектор.

Сохранение деструктурируемого значения

Оригинал деструктурируемой коллекции можно связать с локальным символом, определив имя через опцию `:as` внутри формы деструктуризации:

```
(let [[x _ z :as original-vector] v]
     (conj original-vector (+ x z)))
;= [42 "foo" 99.2 [5 12] 141.2]
```

Здесь с именем `original-vector` будет связано исходное значение `v`. Данная возможность может пригодиться для деструктуризации коллекции, являющейся результатом вызова функции, когда вдобавок к деструктурированным значениям необходимо сохранить ссылку на исходный результат. Если бы это не было возможно, сохранить исходный результат можно было бы так:

```
(let [some-collection (some-function ...)
      [x y z [a b]] some-collection]
  ...выполнить операции с some-collection и ее значениями...)
```

Деструктуризация ассоциативных массивов

Деструктуризация ассоциативных массивов и упорядоченных коллекций концептуально идентичны – мы должны отразить структуру используемой коллекции. Эта разновидность деструктуризации работает с:

- ассоциативными массивами и записями[1];
- любыми коллекциями, реализующими интерфейс `java.util.Map`;
- любыми значениями, поддерживаемыми функцией `get` и позволяющими использовать индексы в качестве ключей:
  - векторы;
  - строки;
  - массивы.

Начнем рассмотрение с деструктуризации ассоциативных массивов:

```
(def m {:a 5 :b 6
        :c [7 8 9]
        :d {:e 10 :f 11}
        "foo" 88
        42 false})
;= #'user/m
(let [{a :a b :b} m]
  (+ a b))
;= 11
```

Здесь значение ключа `:a` в ассоциативном массиве связывается с символом `a`, а значение ключа `:b` – с символом `b`. Возвращаясь к приему визуального упорядочения формы деструктуризации коллекции (в данном случае, ее части), можно снова заметить структурное соответствие:

```
{a  :a b  :b}
{:a 5  :b 6}
```

Обратите внимание, что нет такого требования, чтобы ключи в форме деструктуризации были ключевыми словами (keywords); допускается использовать любые значения:

---

[1] Подробнее о записях рассказывается в разделе «Записи», в главе 6.

```
(let [{f "foo"} m]
  (+ f 12))
;= 100
(let [{v 42} m]
  (if v 1 0))
;= 0
```

Индексы векторов, строк и массивов могут использоваться в качестве ключей при их деструктуризации в ассоциативные массивы[1]. Такая возможность может пригодиться, например, для извлечения лишь нескольких значений из матриц, представленных векторами. Форма деструктуризации ассоциативных массивов для извлечения двух или трех значений из матрицы 3×3 выглядит намного короче, чем полная форма деструктуризации упорядоченной коллекции с указанием девяти элементов:

```
(let [{x 3 y 8} [12 0 0 -18 44 6 0 0 1]]
  (+ x y))
;= -17
```

По аналогии с формами деструктуризации упорядоченных коллекций, формы деструктуризации ассоциативных массивов так же могут быть составными:

```
(let [{{e :e} :d} m]
  (* 2 e))
;= 20
```

Внешняя форма деструктуризации ассоциативного массива – `{{e :e} :d}` – применяется к верхнеуровневой исходной коллекции `m` для извлечения значения, соответствующего ключу `:d`. Внутренняя форма деструктуризации – `{e :e}` – применяется к значению по ключу `:d` для извлечения значения по вложенному ключу `:e`.

Но самой «убойной» является возможность объединения обеих разновидностей деструктуризации, упорядоченных коллекций и

---

[1] Это обусловлено полиморфным поведением функции `get`, которая отыскивает значения в коллекциях по указанным ключам. В данном случае, при поиске значений в упорядоченных коллекциях, в качестве ключей используются индексы. Подробнее о функции `get` рассказывается в разделе «Ассоциативные коллекции» в главе 3.

ассоциативных массивов, позволяющая эффективно извлекать необходимые значения из имеющихся коллекций:

```
(let [{[x _ y] :c} m]
  (+ x y))
;= 16
(def map-in-vector ["James" {:birthday (java.util.Date. 73 1 6)}])
;= #'user/map-in-vector
(let [[name {bd :birthday}] map-in-vector]
  (str name " was born on " bd))
;= "James was born on Thu Feb 06 00:00:00 EST 1973"
```

Механизм деструктуризации ассоциативных массивов обладает также рядом дополнительных возможностей.

Сохранение деструктурируемого значения. Как и при деструктуризации упорядоченных коллекций, добавление пары с ключом `:as` позволяет сохранить ссылку на исходную коллекцию, которую затем можно использовать как любую другую `let`-связку:

```
(let [{r1 :x r2 :y :as randoms}
      (zipmap [:x :y :z] (repeatedly (partial rand-int 10)))]
  (assoc randoms :sum (+ r1 r2)))
;= {:sum 17, :z 3, :y 8, :x 9}
```

Значения по умолчанию. С помощью конструкции `:or` можно определить ассоциативный массив с парами ключ/значение по умолчанию. Если ключ, указанный в форме деструктуризации, отсутствует в исходной коллекции, с символом будет связано значение по умолчанию:

```
(let [{k :unknown x :a
       :or {k 50}} m]
  (+ k x))
;= 55
```

Альтернативой механизму значений по умолчанию может служить либо слияние, либо последовательная проверка и замена пустых (`nil`) элементов. А это может быть неудобно в случае со сложными структурами:

```
(let [{k :unknown x :a} m
       k (or k 50)]
  (+ k x))
;= 55
```

Кроме того, и в отличие от предыдущего примера, конструкция `:or` различает пустое и ложное значения (`nil` и `false`):

```
(let [{opt1 :option} {:option false}
      opt1 (or opt1 true)
      {opt2 :option :or {opt2 true}} {:option false}]
  {:opt1 opt1 :opt2 opt2})
;= {:opt1 true, :opt2 false}
```

Связывание значений с символами, имена которых совпадают с ключами. Часто в качестве ключей в ассоциативных массивах используются устоявшиеся имена, и нередко бывает желательно связать значения по этим ключам с одноименными символами в области видимости формы `let`. Однако, форма деструктуризации таких ассоциативных массивов может содержать множество повторяющихся имен:

```
(def chas {:name "Chas" :age 31 :location "Massachusetts"})
;= #'user/chas
(let [{name :name age :age location :location} chas]
  (format "%s is %s years old and lives in %s." name age location))
;= "Chas is 31 years old and lives in Massachusetts."
```

Необходимость дважды вводить каждый ключ противоречит духу лаконичности. В таких случаях можно использовать конструкции `:keys`, `:strs` и `:syms` для определения ключей (keywords), строк и символов (соответственно) в исходном ассоциативном массиве и имен, которым должны быть присвоены соответствующие значения в форме `let`. В примере ассоциативного массива выше в качестве ключей используются ключевые слова, поэтому можно воспользоваться конструкцией `:keys`:

```
(let [{:keys [name age location]} chas]
  (format "%s is %s years old and lives in %s." name age location))
;= "Chas is 31 years old and lives in Massachusetts."
```

... и использовать конструкцию `:strs` или `:syms`, если известно, что в качестве ключей в исходной коллекции используются строки или символы:

```
(def brian {"name" "Brian" "age" 31 "location" "British Columbia"})
;= #'user/brian
(let [{:strs [name age location]} brian]
```

```
  (format "%s is %s years old and lives in %s." name age location))
;= "Brian is 31 years old and lives in British Columbia."

(def christophe {'name "Christophe" 'age 33 'location "Rhône-Alpes"})
;= #'user/christophe
(let [{:syms [name age location]} christophe]
  (format "%s is %s years old and lives in %s." name age location))
;= "Christophe is 31 years old and lives in Rhône-Alpes."
```

На практике конструкция `:keys` используется гораздо чаще, чем конструкции `:strs` или `:syms`; ключи, представленные ключевыми словами, чаще всего используются в ассоциативных массивах, записях и для представления именованных аргументов (см. ниже).

Извлечение оставшихся пар ключ/значение из коллекции. Мы уже видели как «остальные» значения в формах деструктуризации упорядоченных коллекций могут быть сохранены в последовательность `rest`. Формы деструктуризации упорядоченных коллекций и ассоциативных массивов могут использоваться для работы с любыми структурами. Ниже приводится простой случай вектора, содержащего несколько позиционных значений, за которыми следует ассоциативный массив:

```
(def user-info ["robert8990" 2011 :name "Bob" :city "Boston"])
;= #'user/user-info
```

Данные с подобной организацией встречаются довольно часто, но их обработка редко отличается элегантностью. В Clojure вполне допустимо использовать разбор таких данных «вручную»:

```
(let [[username account-year & extra-info] user-info      ❶
      {:keys [name city]} (apply hash-map extra-info)]    ❷
    (format "%s is in %s" name city))
;= "Bob is in Boston"
```

❶ Исходный вектор деструктуризуется на позиционные элементы и на последовательность, содержащую «оставшуюся часть».

❷ Эта оставшаяся часть, состоящая из пар ключ/значение, используется как основа для создания нового ассоциативного массива, который затем деструктуризуется в соответствии с требованиями.

Однако, «допустимое» не значит лучшее, учитывая распространенность последовательностей пар ключ/значение в программировании. Более удачным решением является специальный механизм

форм деструктуризации: применение деструктуризации ассоциативного массива к оставшейся последовательности. Если оставшаяся последовательность содержит четное число элементов – соответствует семантике ассоциативных массивов (пар ключ/значение) – к ней можно применить операцию деструктуризации ассоциативного массива:

```
(let [[username account-year & {:keys [name city]}] user-info]
  (format "%s is in %s" name city))
;= "Bob is in Boston"
```

Такая форма записи выглядит много проще, чем извлечение данных вручную с созданием промежуточного ассоциативного массива из оставшейся последовательности и последующей его деструктуризации. Кроме того, данный механизм лежит в основе реализации необязательных именованных аргументов функций, описываемых в разделе «Именованные аргументы» (ниже).

Создание функций: fn

Функции в языке Clojure являются *сущностями первого порядка* (first-class values)[1]. А создание их возлагается на специальную форму `fn`, которая также заключает в себе семантику форм `let` и `do`.

Ниже представлена простая функция, прибавляющая 10 к значению аргумента:

```
(fn [x]          ❶
    (+ 10 x))    ❷
```

❶ Форма `fn` принимает вектор привязок в стиле `let`, определяющий имена и количество аргументов, принимаемых функцией; к каждому аргументу применяются те же формы деструктуризации, что обсуждались в разделе «Деструктуризация (let, часть 2)».

[1] Термин «first-class values» не имеет устоявшегося перевода на русский язык. В Интернете часто можно встретить такие толкования, как «объекты первого рода», «объекты первого класса» и даже «первоклассные значения». Однако, чтобы не вносить путаницу в умы тех, кто имеет опыт объектно-ориентированного программирования, и подчеркнуть, что речь идет не о типах, а о свойстве функциональных языков программирования, было решено использовать толкование «сущности первого порядка». – *Прим. перев.*

❷ Формы, следующие за вектором привязок, образуют *тело* функции. Тело неявно заключается в форму `do`, благодаря чему тело любой функции может содержать любое количество форм. Как и в форме `do`, последняя форма в теле функции определяет возвращаемое значение.

Аргументы функции соотносятся с именами или формами деструктуризации согласно их порядку в форме вызова. То есть, в следующем вызове:

```
((fn [x] (+ 10 x)) 8)
;= 18
```

8 является единственным аргументом функции, который соответствует имени `x` в теле функции. Это делает вызов функции эквивалентным следующей форме `let`:

```
(let [x 8]
  (+ 10 x))
```

Можно определить функцию, принимающую несколько аргументов:

```
((fn [x y z] (+ x y z))
  3 4 12)
;= 19
```

В этом случае вызов функции эквивалентен следующей форме `let`:

```
(let [x 3
      y 4
      z 12]
  (+ x y z))
```

Имеется также возможность создавать функции, *работающие по-разному, в зависимости от количества аргументов*; в следующем примере мы сохранили функцию в переменную, чтобы ее можно было вызывать многократно, обращаясь к имени переменной:

```
(def strange-adder (fn adder-self-reference
                     ([x] (adder-self-reference x 1))
                     ([x y] (+ x y))))
;= #'user/strange-adder
```

```
(strange-adder 10)
;= 11
(strange-adder 10 50)
;= 60
```

При определении функции с несколькими арностями (arities), вектор привязок и тело функции для каждой арности должны быть заключены в круглые скобки. При вызове функции соответствующая арность выбирается исходя из количества переданных аргументов.

Обратите внимание на необязательное имя `adder-self-reference`, передаваемое функции в последнем примере. Этот необязательный первый аргумент в форме `fn` может использоваться в функции для ссылки на саму функцию. В данном случае арность, принимающая единственный аргумент, может вызывать арность, принимающую два аргумента, добавляя второй аргумент по умолчанию, не требуя передачи вмещающей переменной.

Примечание. Реализация взаимно-рекурсивных функций с применением формы letfn. Именованные функции (как `adder-self-reference` в примере выше) позволяют легко создавать рекурсивные функции. Однако иногда бывает необходимо определить взаимно-рекурсивные функции.

В таких редких случаях применяется специальная форма `letfn`, позволяющая одновременно определить несколько именованных функций, каждая из которых будет знать о существовании друг друга. Взгляните на примитивную реализацию функций `odd?` и `even?`:

```
(letfn [(odd? [n]
          (even? (dec n)))
        (even? [n]
          (or (zero? n)
            (odd? (dec n))))]          ❶
  (odd? 11))
;= true
```

❶ Вектор привязки включает несколько определений `fn`, но без символа `fn`.

defn основывается на fn. Мы уже видели примеры использования `defn` выше. В действительности `defn` является макросом, инкапсулирующим функциональность форм `def` и `fn`, и обеспечивающим короткий способ определения именованных функций и их регистрацию в указанном пространстве имен. Например, следующие два определения эквивалентны:

```
(def strange-adder (fn strange-adder
                     ([x] (strange-adder x 1))
                     ([x y] (+ x y))))

(defn strange-adder
  ([x] (strange-adder x 1))
  ([x y] (+ x y))))
```

С помощью этой формы можно определять и функции с одной арностью, при этом ликвидируется лишняя пара скобок. Например, следующие два определения эквивалентны:

```
(def redundant-adder (fn redundant-adder
                       [x y z]
                       (+ x y z)))

(defn redundant-adder
  [x y z]
  (+ x y z))
```

В оставшейся части этого раздела мы часто будем использовать форму `defn` вместо формы `fn`, просто потому, что вызов именованных функций читается проще, чем одновременное определение функции и ее вызов.

Деструктуризация аргументов функций

Форма `defn` поддерживает деструктуризацию аргументов функции, благодаря использованию формы `let` для связывания аргументов функции с именами в области видимости тела функции. Полный перечень возможностей различных форм деструктуризации приводится в обсуждении выше. Здесь же мы рассмотрим лишь пару идиом деструктуризации, которые часто используются при определении функций.

Функции с переменным числом аргументов. При необходимости функции могут собирать все дополнительные аргументы в последовательность; здесь используется тот же механизм, что и в деструктуризации упорядоченных коллекций, когда «остальные» значения сохраняются в последовательности. Такие функции называются *функциями с переменным числом аргументов* (variadic). Ниже приводится пример функции, имеющий один именованный позиционный аргумент и собирающую оставшиеся аргументы в последовательность:

```
(defn concat-rest
  [x & rest]
  (apply str (butlast rest)))
;= #'user/concat-rest
(concat-rest 0 1 2 3 4)
;= "123"
```

Последовательность, формируемая из оставшихся аргументов, может быть подвергнута деструктуризации, как любая другая последовательность. Следующий пример демонстрирует использование деструктуризации для оставшихся аргументов, чтобы функция работала будто в ней определена логика для нулевой арности:

```
(defn make-user
  [& [user-id]]
  {:user-id (or user-id
                (str (java.util.UUID/randomUUID)))})
;= #'user/make-user
(make-user)
;= {:user-id "ef165515-6d6f-49d6-bd32-25eeb024d0b4"}
(make-user "Bobby")
;= {:user-id "Bobby"}
```

Именованные аргументы. Часто бывает необходимо определить функцию, принимающую множество аргументов, где одни могут быть необязательными, а другие – иметь значения по умолчанию. И так же не желательно использовать жесткий порядок аргументов при вызове функций[1].

Форма `fn` (и, соответственно, `defn`) решает эти проблемы с помощью *именованных аргументов* – идиомы, основанной на механизме деструктуризации ассоциативного массива формы `let`, применяемой к последовательности «остальных» значений. Именованные аргументы представляют собой пары ключевых слов и значений, добавляемые при вызове функций в конец списка строго позиционируемых аргументов. Если функция определена так, что способна принимать именованные аргументы, эти пары будут собраны в ассоциативный массив и обработаны формой деструктуризации, которая

[1] В языке Python поддерживается подобный механизм, где аргументы могут иметь значения по умолчанию, а так же имена, чтобы их можно было указывать при вызове в произвольном порядке.

располагается в векторе привязок на месте для последовательности «остальных» аргументов:

```
(defn make-user
  [username & {:keys [email join-date]                    ❶
               :or {join-date (java.util.Date.)}}]        ❷
  {:username username
   :join-date join-date
   :email email
   ;; 2.592e9 -> один месяц в мсек
   :exp-date (java.util.Date. (long (+ 2.592e9 (.getTime join-date))))})
;= #'user/make-user
(make-user "Bobby")                                       ❸
;= {:username "Bobby", :join-date #<Date Mon Jan 09 16:56:16 EST 2012>,
;= :email nil, :exp-date #<Date Wed Feb 08 16:56:16 EST 2012>}
(make-user "Bobby"                                        ❹
  :join-date (java.util.Date. 111 0 1)
  :email "bobby@example.com")
;= {:username "Bobby", :join-date #<Date Sun Jan 01 00:00:00 EST 2011>,
;= :email "bobby@example.com", :exp-date #<Date Tue Jan 31 00:00:00 EST 2011>}
```

❶ Функция `make-user` требует один обязательный аргумент – имя пользователя. Остальные аргументы, которые, как предполагается, передаются в виде пар имя/значение, будут собраны в ассоциативный массив и затем обработаны с применением формы деструктуризации, следующей за `&`.

❷ В форме деструктуризации ассоциативного массива определено значение по умолчанию для имени `join-date`.

❸ Вызов функции `make-user` с единственным аргументом возвращает ассоциативный массив с информацией о пользователе, где поля `join-date` и `expiration-date` получают значение по умолчанию, а поле email получает значение `nil`.

❹ При передаче функции `make-user` дополнительных аргументов, они интерпретируются с применением формы деструктуризации, без учета порядка их следования.

Примечание. Так как поддержка именованных аргументов основана на механизме деструктуризации ассоциативных массивов формы `let`, ничто не мешает делать деструктуризацию «остальных» аргументов как ассоциативного массива с использованием на месте ключей не только ключевых слов, но и строк, чисел и даже коллекций. Например:

```
(defn foo
     [& {k ["m" 9]}]
     (inc k))
```

```
;= #'user/foo
(foo [":m" 9] 19)
;= 20
```

`[":m" 9]` интерпретируется здесь, как имя «именованного» аргумента. Однако в нашей практике мы никогда не встречали использование других типов для ключей в именованных аргументах. Ключевые слова – это наиболее распространенный тип ключей, поэтому для описания идиомы мы используем термин «именованные аргументы» (keyword arguments).

Пред- и постусловия. Форма `fn` обеспечивает поддержку пред- и постусловий для реализации проверок аргументов и возвращаемых значений. Это весьма ценная возможность, которая может использоваться для тестирования и соблюдения инвариантов функций (enforcing function invariants); подробнее эти особенности будут обсуждаться в разделе «Предусловия и постусловия» в главе 13.

Литералы функций

Литералы функций уже упоминались в разделе «Прочий синтаксический сахар механизма чтения». Литералы функций в языке Clojure являются эквивалентами блоков в языке Ruby и `lambda`-функций в языке Python. Они обеспечивают лаконичный синтаксис для определения простых анонимных функций.

Например, следующие анонимные функции эквивалентны:

```
(fn [x y] (Math/pow x y))

#(Math/pow %1 %2)
```

Последнее из этих определений – всего лишь синтаксический сахар механизма чтения, которое преобразуется в первое. В этом можно убедиться, проверив результат «чтения» этого кода[1]:

```
(read-string "#(Math/pow %1 %2)")
;= (fn* [p1__285# p2__286#] (Math/pow p1__285# p2__286#))
```

Далее перечисляются различия между формой `fn` и литералом функции:

Отсутствие явной формы do. «Обычная» форма `fn` (и все ее производные) неявно заворачивает тело функции в форму `do`, как уже

[1] Так как во втором случае имена аргументов отсутствуют, литерал функции генерирует уникальные символы для каждого из аргументов; в данном случае `p1__285#` и `p2__286#`.

говорилось в разделе «Создание функций: fn». Это позволяет, например, создавать такие определения:

```
(fn [x y]
  (println (str x \^ y))
  (Math/pow x y))
```

Эквивалентный литерал функции требует явного использования формы `do`:

```
#(do (println (str %1 \^ %2))
   (Math/pow %1 %2))
```

Арность и ее аргументы указываются с использованием неименованных позиционных символов. В примере выше символы `x` и `y` определяют арность и имена аргументов функции, передаваемых ей при вызове. В литералах, напротив, используются неименованные позиционные символы `%`, где `%1` обозначает первый аргумент, `%2` — второй аргумент, и так далее. Кроме того, наибольший позиционный символ определяет арность функции. Если потребуется определить литерал функции, принимающей четыре аргумента, достаточно просто сослаться на символ `%4` в его определении.

Существуют две дополнительные особенности, касающиеся определения аргументов в литералах функций:

1. Литералы функций с единственным аргументом настолько часто используются в практике, что предусматривается возможность ссылаться на первый аргумент используя простой символ `%`. То есть, определение `#(Math/pow % %2)` эквивалентно определению `#(Math/pow %1 %2)`. В общем случае рекомендуется использовать более короткую форму записи.
2. Допускается определять функции с переменным числом аргументов[1] и ссылаться на остальные аргументы в теле функции с использованием символа `%&`. Например, следующие определения функций эквивалентны:

```
(fn [x & rest]
     (- x (apply + rest)))

#(- % (apply + %&))
```

[2] См. подраздел «Функции с переменным числом аргументов» в разделе «Деструктуризация аргументов функции» выше.

Литералы функций не могут быть вложенными. Если следующее определение является допустимым:

```
(fn [x]
  (fn [y]
    (+ x y)))
```

то это определение – нет:

```
#(#(+ % %))
;= #<IllegalStateException java.lang.IllegalStateException:
;= Nested #()s are not allowed>
```

Во-первых, литералы функций предназначены для создания простых и кратких выражений, а возможность определения вложенных литералов функций могла бы превратить программный код в кошмар для тех, кто будет его читать. Во-вторых, невозможно однозначно сказать, на первый аргумент какого литерала ссылается символ `%`.

Выполнение по условию: if

`if` – единственный в языке Clojure простой условный оператор. Он имеет очень простой синтаксис: если значение первого выражения в форме `if` оценивается как *логическая* истина, результатом всей формы `if` становится значение второго выражения. В противном случае, результатом формы `if` становится значение третьего выражения, если оно присутствует. Второе и третье выражение вычисляются только в случае необходимости.

В языке Clojure истинным в логическом контексте считается любое значение, отличное от `nil` и `false`:

```
(if "hi" \t)
;= \t
(if 42 \t)
;= \t
(if nil "unevaluated" \f)
;= \f
(if false "unevaluated" \f)
;= \f
(if (not true) \t)
;= nil
```

Обратите внимание, что если условное выражение оценивается как логическая ложь и третье выражение отсутствует, в качестве результата формы `if` возвращается значение `nil`[1].

На основе формы `if` построено несколько дополнительных форм:

- `when`, лучше подходит для случаев, когда при невыполнении условия должно возвращаться значение `nil` (или не должно предприниматься никаких действий);
- `cond` – напоминает конструкцию `else if` в Java и Ruby, и `elif` в Python – обеспечивает более лаконичную форму записи нескольких условий и нескольких выражений, вычисляемых при выполнении условия;
- `if-let` и `when-let`, являются производными от объединения формы `let` с формой `if` или `when`, соответственно: если значение условного выражения является истинным, с локальным символом связывается значение второго (`then`) выражения.

Внимание. В языке Clojure имеются предикаты `true?` и `false?`, но они никак не связаны с условным оператором `if`. Например:

```
(true? "string")
;= false
(if "string" \t \f)
;= \t
```

`true?` и `false?` проверяют на равенство логическим значениям `true` и `false`, а не истинность в логическом контексте, как оператор `if`, который, фактически, эквивалентен конструкции `(or (not (nil? x)) (true? x))` для произвольного значения `x`.

Организация циклов: loop и recur

В языке Clojure имеется несколько удобных конструкций для организации циклического выполнения кода, включая `doseq` и `dotimes`, основанные на форме `recur`. Форма `recur` передает управление в ближайший *заголовок цикла* (не потребляя пространство на стеке), который определяется либо формой `loop`, либо функцией. Рассмотрим простейший цикл, выполняющий обратный отсчет:

```
(loop [x 5]                ❶
  (if (neg? x)
    x                      ❷
    (recur (dec x))))      ❸
;= -1
```

---

[1] Для таких случаев лучше подходит форма `when`.

❶ Форма `loop` выполняет связывание, неявно используя форму `let`, поэтому она принимает вектор с именами и начальными значениями.
❷ Если последнее выражение в форме `loop` состоит из значения, оно принимается в качестве результата всей формы. В данном примере, когда `x` станет меньше нуля, форма `loop` вернет значение `x`.
❸ Форма `recur` передаст управление ближайшему заголовку цикла и переустановит локальные привязки в соответствии со своим аргументом. В этом примере управление будет передано в начало формы `loop`, при этом символ `x` будет связан со значением `(dec x)`.

Функции так же могут играть роль заголовка цикла. В этом случае форма `recur` выполняет повторную привязку параметров функции, используя значения своих аргументов:

```
(defn countdown
  [x]
  (if (zero? x)
    :blastoff!
    (do (println x)
        (recur (dec x)))))
;= #'user/countdown
(countdown 5)
; 5
; 4
; 3
; 2
; 1
;= :blastoff!
```

Правила использования формы recur. Форма `recur` является весьма низкоуровневым механизмом организации циклов и рекурсии, потребность в котором возникает редко:

- когда это возможно, используйте высокоуровневые формы организации циклов `doseq` и `dotimes`, имеющиеся в стандартной библиотеке Clojure;
- когда требуется выполнить обход элементов коллекции или последовательности, предпочтительнее использовать функциональные операции, такие как `map`, `reduce`, `for`, и другие.

Применение формы `recur` не влечет за собой расходование пространства на стеке (благодаря чему исключается возможность переполнения стека), поэтому форма `recur` играет важную роль в реали-

зации некоторых рекурсивных алгоритмов. Кроме того, она позволяет работать с числами без преобразования их в объектные типы, что делает ее полезным инструментом для реализации многих математических вычислений и операций обработки данных. Практический пример применения формы `recur` в подобных целях можно найти в разделе «Визуализация множества Мандельброта в Clojure» в главе 11.

Напоследок хотелось бы упомянуть, что в некоторых случаях может понадобиться организовать более сложные манипуляции с набором коллекций, которые трудно реализовать в виде последовательности функциональных операций (`map`, `reduce` и других) или такая реализация оказывается неэффективной. В таких ситуациях на помощь может прийти форма `recur` (и иногда форма `loop`, для установки промежуточных заголовков циклов).

Ссылки на переменные: var

Символы, являющиеся именами переменных, интерполируются в их значения:

```
(def x 5)
;= #'user/x
x
;= 5
```

Однако иногда бывает необходимо иметь ссылку на переменную, а не само значение. Получить такую ссылку можно с помощью специальной формы `var`:

```
(var x)
;= #'user/x
```

Вы уже неоднократно видели, как выводятся ссылки в REPL: последовательность знаков `#'`, за которой следует символ (symbol). Этот синтаксис преобразуется механизмом чтения в ссылку на var:

```
#'x
;= #'user/x
```

Подробнее о переменных рассказывается в разделе «Переменные», в главе 4.

Взаимодействие с Java: . и new

Все взаимодействия с Java – создание экземпляров классов, вызовы методов и обращение к полям объектов – выполняются посредством специальных форм `new` и `.` (точка). Кроме того, механизм чтения в языке Clojure предоставляет дополнительные синтаксические конструкции, построенные поверх этих элементарных форм, упрощающие и делающие эти взаимодействия с Java синтаксически похожими на взаимодействие с кодом на Clojure. Хотя, редко можно увидеть прямое использование форм `.` (точка) и `new`, но вы наверняка столкнетесь с ними в своей практике.

Таблица 1.3. Формы взаимодействия с Java и их эквиваленты

Операция	Код на Java	Синтаксический сахар Clojure	Эквивалент на основе специальной формы
Создание экземпляра	`new java.util.ArrayList(100)`	`(java.util.ArrayList.100)`	`(new java.util.ArrayList 100)`
Вызов статического метода	`Math.pow(2, 10)`	`(Math/pow 2 10)`	`(. Math pow 2 10)`
Вызов метода экземпляра	`"hello".substring(1, 3)`	`(.substring "hello" 1 3)`	`(. "hello" substring 1 3)`
Доступ к статическому полю	`Integer.MAX_VALUE`	`Integer/MAX_VALUE`	`(. Integer MAX_VALUE)`
Доступ к полю экземпляра	`someObject.someField`	`(.someField someobject)`	`(. someobject somefield)`

Синтаксический сахар, представленный в табл. 1.3, является идиоматическим для языка Clojure и является более предпочтительным, нежели непосредственное использование специальных форм `.` и `new`. Более подробно о взаимодействии с Java рассказывается в главе 9.

Обработка исключений: try и throw

Эти специальные формы позволяют использовать механизм исключений Java из программ на языке Clojure. Подробнее они рассматриваются в разделе «Обработка исключений и ошибок», в главе 9.

Специализированная операция set!

Хотя в языке Clojure основной упор делается на использовании неизменяемых значений и структур данных, однако бывают задачи, требующие непосредственной модификации состояния. Чаще всего для этих целей используются методы записи (setters) свойств и другие методы объектов Java, изменяющие их состояние. Также в языке Clojure имеется форма `set!`, которую можно использовать:

- с целью установки переменных, локальных для потоков выполнения, как обсуждается в разделе «Динамическая область видимости», в главе 4;
- для изменения значений полей Java-объектов, как демонстрируется в разделе «Доступ к полям объектов», в главе 9;
- для изменения значений изменяемых полей, объявленных с помощью `deftype`; см. раздел «Типы» в главе 6.

Примитивы блокировок: monitor-enter и monitor-exit

Упомянутые в заголовке примитивы блокировок позволяют выполнять синхронизацию с мониторами соответствующих Java-объектов. Вам никогда не придется использовать эти специальные формы, потому что существует более удобный макрос `locking`, гарантирующий своевременное приобретение и освобождение монитора объекта. Более подробное описание приводится в разделе «Блокировка», в главе 4.

Все вместе

В процессе нашего знакомства с языком Clojure мы не раз возвращались к примеру 1.2:

```
(defn average
  [numbers]
  (/ (apply + numbers) (count numbers)))
```

В разделе «Гомоиконность» мы узнали, что это выражение является каноническим представление структуры данных на языке Clojure. В начале раздела «Выражения, операторы, синтаксис и очередность» мы установили, что списки интерпретируются как вызовы, где элемент в позиции функции интерпретируется как опера-

тор. После знакомства с пространствами имен, мы увидели в разделе «Интерпретация символов», как выполняется интерпретация символов в этих структурах данных на этапе выполнения. Теперь, познакомившись со специальными формами, особенно такими, как `def` и `fn`, мы получили полное представление, что происходит, когда вычисляются выражения (будь это в интерактивной оболочке REPL или на стороне отдельного приложения).

Форма `defn` является более лаконичным вариантом для:

```
(def average (fn average
               [numbers]
               (/ (apply + numbers) (count numbers))))
```

То есть, форма `fn` создает функцию `average` (как описывается в разделе «Создание функций: fn», выше, первый аргумент формы `fn` (`average`) является ссылкой на саму функцию, благодаря чему функция может рекурсивно вызывать саму себя без необходимости обращаться к внешней переменной), а форма `def` регистрирует ее как переменную с именем `average` в текущем пространстве имен.

eval

Вся семантика вычислений, обсуждавшаяся выше, инкапсулирована в функции `eval`, которая вычисляет единственный аргумент формы. Мы можем очень ясно увидеть, что, например, скаляры и другие литералы интерпретируются как описываемые ими значения:

```
(eval :foo)
;= :foo
(eval [1 2 3])
;= [1 2 3]
(eval "text")
;= "text"
```

...а список преобразуется в результат соответствующего вызова:

```
(eval '(average [60 80 100 400]))
;= 160
```

Внимание. Несмотря на то, что функция `eval` включает всю семнатику языка Clojure, сама она крайне редко используется в программах. Она обеспечивает непревзойденную гибкость, позволяя интерпретировать

любые синтаксически допустимые структуры данных на языке Clojure, которая просто не нужна в большинстве случаев. Проще говоря, если вы используете функцию `eval` в прикладном коде, вероятнее всего вы создадите себе много проблем в процессе работы.
Большинство задач, где можно было бы применить функцию `eval`, гораздо проще решаются с помощью макросов, которые мы будем исследовать в главе 5.

Используя знания, полученные к данному моменту, мы легко можем *реализовать* собственную версию Clojure REPL. Помня, что получение значений на языке Clojure из их текстового представления обеспечивает функция `read` (или `read-string`):

```
(eval (read-string "(average [60 80 100 400])"))
;= 160
```

...можно сконструировать цикл чтения команд, применив форму `recur` внутри функции (с таким же успехом можно было бы использовать форму `loop`). Достаточно добавить вызовы функций для вывода результатов и строки приглашения к вводу, и вы получите полнофункциональную версию REPL:

Пример 1.4. Простейшая реализация Clojure REPL

```
(defn embedded-repl
  "A naive Clojure REPL implementation. Enter ':quit'
   to exit."
  []
  (print (str (ns-name *ns*) ">>> "))
  (flush)
  (let [expr (read)
        value (eval expr)]
    (when (not= :quit value)
      (println value)
      (recur))))

(embedded-repl)
; user>>> (defn average2
; [numbers]
; (/ (apply + numbers) (count numbers)))
; #'user/average2
; user>>> (average2 [3 7 5])
; 5
; user>>> :quit
;= nil
```

Данная реализация REPL страдает некоторыми недостатками, например, она не перехватывает исключения, которые могут произойти в работе, но это всего лишь прототип[1].

Это лишь начало

В этой главе мы исследовали лишь суть языка Clojure: базовые операции (специальные формы), взаимозаменяемость кода и данных и познакомились с интерактивной разработкой. На этой концептуальной основе подкрепленной возможностями JVM, язык Clojure предоставляет неизменяемые структуры данных; примитивы конкурентного программирования с четкой и понятной семантикой; макросы; и многое, многое другое.

Мы будем помогать вам получать новые знания на протяжении всей книги, и надеемся, что Clojure станет языком программирования для ваших повседневных задач благодаря практическим заданиям в четвертой части.

Ниже перечислены некоторые ресурсы, которые могут вам пригодиться:

- документация с описанием базового API доступна по адресу http://clojure.github.com/clojure;
- обширный репозиторий с документацией и примерами, сопровождаемый сообществом, доступен по адресу http://clojuredocs.org/;
- главный список рассылки Clojure, доступный по адресу http://groups.google.com/group/clojure, и IRC-канал `#clojure` на сайте Freenode[2] являются отличными способами получить квалифицированную помощь по языку Clojure, независимо от уровня вашей подготовки;
- сайт, сопутствующий данной книге, http://clojurebook.com, который будет пополняться материалами, призванными помочь в изучении и эффективном использовании Clojure.

Вы готовы сделать следующий шаг?

---

[1] В действительности интерактивная оболочка REPL также написана на языке Clojure. Ее реализацию можно найти в пространстве имен `clojure.main`, где каждый желающий сможет посмотреть, как устроен этот чудесный инструмент внутри.

[2] Если у вас не установлен локальный IRC-клиент, можете воспользоваться веб-интерфейсом: http://webchat.freenode.net/?channels=#clojure.

Часть I
ФУНКЦИОНАЛЬНОЕ И КОНКУРЕНТНОЕ ПРОГРАММИРОВАНИЕ

Глава 2. Функциональное программирование

Словосочетание «функциональное программирование» (ФП) является одним из тех аморфных понятий в разработке ПО, которые разные люди понимают по-разному. Несмотря на все полутона и оттенки, имеющиеся в понятии ФП, можно уверенно утверждать, что Clojure является языком функционального программирования, и эта его особенность является причиной его привлекательности.

Далее мы:

1. Расскажем, что такое «функциональное программирование».
2. Объясним, зачем оно необходимо.
3. Обсудим особенности Clojure, которые делают его языком функционального программирования.

Попутно мы надеемся показать, что ФП – и Clojure, как язык ФП в частности, – представляет не только академический интерес и способно ваши расширить возможности в проектировании и разработке ПО, также как структурное и объектно-ориентированное программирование.

Если вы уже знакомы с функциональным программированием (например, благодаря использованию языка Ruby или JavaScript, или даже таких функциональных языков, как Scala, F# или Haskell), большая часть из того, о чем будет рассказываться ниже, покажется вам хорошо знакомой, но эти сведения совершенно необходимы, чтобы вы могли понять функциональные черты Clojure.

Если вы вообще не знакомы с функциональным программированием или испытываете скепсис относительно его практической ценности, мы настоятельно рекомендуем продолжить чтение – это стоит ваших сил и времени[1]. В главе 1 уже говорилось, что *язык Clojure*

---

[1] После знакомства со сведениями в этой главе, вы можете обнаружить, что страница в Википедии, посвященная функциональному программированию (http://ru.wikipedia.org/wiki/Функциональное_программирование) является удивительно хорошим трамплином для погружения в родственные темы.

требует вложить время и силы в его изучение, но все затраты окупятся сторицей. Так же, как вам, возможно, пришлось приложить силы для освоения объектно-ориентированного программирования, или шаблонов (generics) в языке Java, или языка Ruby, точно так же придется приложить некоторые усилия, чтобы освоить приемы ФП и, соответственно, язык Clojure. Однако взамен вы получите не только «новое мышление», но еще и множество инструментов и практических приемов, пригодных дл использования в повседневной работе[1].

Что подразумевается под термином «Функциональное программирование»?

Функциональное программирование – это обобщающее понятие, охватывающее множество различных языковых конструкций и механизмов, которые в разных языках реализуются по-разному. В Clojure термин «функциональное программирование» означает:

- предпочтительное положение неизменяемых *значений*, в том числе:
  - использование неизменяемых структур данных, удовлетворяющих простым абстракциям, вместо свободно изменяемого состояния;
  - интерпретация функций как значений, что позволяет создавать *функции высшего порядка*;
- предпочтительное положение декларативной обработки данных перед императивными структурами ветвления и управления циклами;
- естественное пошаговое конструирование функций, функций высшего порядка и неизменяемых структур данных, решение сложных задач за счет использования высокоуровневых абстракций.

Все это является основой для множества улучшенных возможностей языка Clojure, о которых вы могли слышать. Особенно хо-

---

[1] Обратите внимание, что приемы функционального программирования *можно использовать* даже в таких языках, как Java, которые не поощряют (а иногда даже активно противодействуют) стилю ФП. Делать это намного проще, если в вашем распоряжении имеются неизменяемые структуры данных и реализации основных примитивов ФП, подобные тем, что предоставляются библиотеками из проектов Google Guava (https://code.google.com/p/guava-libraries/) и Functional Java (http://functionaljava.org).

телось бы отметить фантастическую поддержку многозадачности и параллельного программирования в языке Clojure, а также наличие семантики управления идентичностями (identities) и состояниями, более подробно описываемых в главе 4.

О важности значений

Понятие *состояние программы* имеет достаточно широкое толкование и длинную историю, но в общем случае под ним подразумеваются все скаляры и структуры данных, используемые для представления сущностей внутри приложения, а также всех связей приложения с внешним миром (таких как открытые файлы, сокеты и прочее). Характер языка программирования в значительной степени определяется его отношением к обработке состояния: что позволяет, чему препятствует и что приветствует.

Большинство языков программирования, посредством идиом или явного синтаксиса, поощряет использование *изменяемого* состояния в виде объектов или структур данных. Языки функционального программирования, напротив, поощряют использование *неизменяемых* объектов – называемых *значениями* – для представления состояния программы. Язык Clojure не является исключением в этом отношении.

«Но, минутку!», – можете сказать вы: «Разговор об отказе от изменяемости данных не имеет смысла. Программа *должна взаимодействовать с окружающим миром*, поэтому изменение ее состояния неизбежно.» Вы определенно правы, что любая, мало-мальски полезная программа должна взаимодействовать с окружающим миром, чтобы получать исходные данные и возвращать результаты, но это не препятствует использованию неизменяемых значений. Напротив, чем больше операций с неизменяемыми значениями будет выполняться в программе, тем проще будет анализировать ее поведение, в сравнении с программой, опирающейся на использование изменяемого состояния. На протяжении этой главы нам не раз представится возможность убедиться в правоте этих слов.

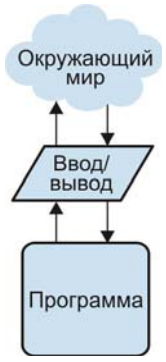

Рис. 2.1. Организация типичной программы

Переход от изменяемого состояния и объектов к неизменяемым значениям для мно-

гих может оказаться непростым делом, но, возможно, вас подбодрит знание, что вы, как программист, уже пользуетесь неизменяемыми значениями в повседневной практике, и что они являются самыми надежными элементами в вашем приложении.

О значениях

Что такое «значения» и чем отличаются неизменяемые значения от изменяемых объектов? Ниже приводится несколько примеров значений:

```
true false 5 14.2 \T "hello" nil
```

Это – стандартные логические, числовые, символьные и строковые значения в JVM, которые также используются в Clojure. Все они являются неизменяемыми, все они являются значениями и всеми ими (или их аналогами в других языках) вы пользуетесь каждый день.

Ключевой особенностью значений является поддержка семантики равенства и сравнения. Например, следующие выражения *всегда* возвращают значение `true`:

```
(= 5 5)

(= 5 (+ 2 3))

(= "boot" (str "bo" "ot"))

(= nil nil)

(let [a 5]
     (do-something-with-a-number a)
     (= a 5))
```

Эквивалентные выражения на языках Java, Python и Ruby так же *всегда* возвращают `true`[1], и этот факт позволяет нам рассуждать об операциях, вовлекающих такие значения. Чтобы понять причину,

[1] Исключение составляет выражение `(= nil nil)`; многие языки имеют весьма ограниченную поддержку сравнения со значением `nil` или `null`, и обычно передача указателя `null` методу `equals` объекта вызывает неприятности. Однако в Clojure значение `nil` является самым обычным значением.

было бы полезно увидеть, что может произойти, если один из таких хорошо знакомых типов потеряет семантику значения.

Сравнение значений изменяемых объектов

Выбор между изменяемыми объектами и неизменяемыми значениями является важным решением, имеющим далеко идущие последствия, даже в самых простых случаях. Однако это решение часто принимается на основе сложившихся привычек, без учета возможных последствий. Поскольку состояние хранится в виде объектов, доступных для изменения, и может изменяться без вашего ведома, их использование при наличии неизменяемых альтернатив может быть опасным.

Это утверждение может показаться преувеличением, особенно для тех, кто использует изменяемые объекты в повседневной практике и не наблюдает особых проблем. Однако, давайте посмотрим, что может произойти, если сделать изменяемым привычное неизменяемое значение, такое как целое число:

Пример 2.1. Реализация изменяемого целого числа на языке Java

```
public class StatefulInteger extends Number {
     private int state;

     public StatefulInteger (int initialState) {
          this.state = initialState;
     }

     public void setInt (int newState) {
          this.state = newState;
     }

     public int intValue () {
          return state;
     }

     public int hashCode () {
          return state;
     }

     public boolean equals (Object obj) {
          return obj instanceof StatefulInteger &&
               state == ((StatefulInteger)obj).state;
     }

     // остальные методы xxxValue() класса java.lang.Number...
}
```

Этот класс в значительной степени идентичен классу `java.lang.Integer`, за исключением отсутствия различных статических методов. Самое важное изменение в этом классе – единственное поле в нем является изменяемым (то есть, оно не объявлено финальным (`final`)) и для него предусмотрен метод записи, `setInt(int)`. Посмотрим, как можно использовать это изменяемое число[1]:

```
(def five (StatefulInteger. 5))          ❶
;= #'user/five
(def six (StatefulInteger. 6))
;= #'user/six
(.intValue five)                         ❷
;= 5
(= five six)                             ❸
;= false
(.setInt five 6)                         ❹
;= nil
(= five six)                             ❺
;= true
```

❶ Создается пара экземпляров класса `StatefulInteger`, один со значением `5`, а второй – со значением `6`.
❷ Проверяется, что объект `five` действительно содержит значение «5»...
❸ ...и `5` не равно `6`.
❹ Изменим значение объекта `five` на `6`, и...
❺ теперь объект `five` стал равен `6`.

Такое положение вещей должно вызывать сильное беспокойство. Мы привыкли к тому, что числа *действительно являются* значениями, причем, не только в техническом смысле. Число `5` всегда должно быть равно `5`, и не должно изменяться по прихоти программиста.

Заключительный пример, иллюстрирующий последствия применения изменяемых объектов в вызовах функций или методов:

```
(defn print-number                       ❶
    [n]
    (println (.intValue n))
    (.setInt n 42))
```

[1] Мы будем работать с этим Java-классом из программного кода на языке Clojure, который обладает богатыми возможностями взаимодействия с Java и JVM. Подробности см. в главе 9.

```
;= #'user/print-number
(print-number six)                    ❷
; 6
;= nil
(= five six)                          ❸
;= false
(= five (StatefulInteger. 42))
;= true
```

❶ Простейшая функция `print-number`, которая якобы должна просто вывести значение заданного числа в `stdout`. Однако, без нашего ведома она изменяет аргумент `StatefulInteger`[1].

❷ Вызов функции `print-number` и передача ей экземпляра `six` класса `StatefulInteger`. Когда функция вернет управление, объект окажется изменен.

❸ Теперь, в противоположность предыдущему примеру, объект `six` более не равен объекту `five`.

В общем случае функция, получающая изменяемые аргументы, как в примере выше, не должна быть вредоносной и необязательно написана некомпетентным программистом. Изменение аргументов внутри функций и методов является распространенным явлением и вам, возможно, уже приходилось сталкиваться (и даже вносить самому!) с ошибками, вызванными изменением внутри метода объекта, переданного в виде аргумента. То же относится и к изменяемым объектам, доступным глобально. Соответственно, никакая документация (которая читается еще реже, чем пишется!) не способна защитить от подобных ситуаций, и такие ловушки, вызванные изменяемостью, являются серьезным основанием для организации глубокого копирования объектов и создания конструкторов копирования.

Если бы числа в вашем языке программирования действовали аналогичным образом, вы могли бы завтра же бросить свою работу и заняться более простым делом. К сожалению, практически все объекты *действуют* именно таким способом.

В Ruby изменяемыми являются даже строки, обычно неизменяемые в других языках. Эта особенность может стать источником самых разнообразных проблем:

---

[1] Мы могли бы реализовать эту функцию как Java-метод. Однако для нашего обсуждения гораздо удобнее использовать функцию на Clojure.

```
>> s = "hello"
=> "hello"
>> s << "*"
=> "hello*"
>> s == "hello"
false
```

Коллекции в языке Ruby, такие как хеши и множества, как бы «замораживают» строки после их добавления в коллекцию. Однако, во всех классах и структурах, создаваемых вами и содержащих строки, необходимо соблюдать меры предосторожности, чтобы предотвратить появление серьезных проблем, относящихся к той же разновидности ошибок, возникающих при использовании изменяемых объектов в качестве ключей в хешах:

```
>> h = {[1, 2] => 3}          ❶
=> {[1, 2]=>3}
>> h[[1,2]]                   ❷
=> 3
>> h.keys[0] << 3             ❸
=> [1, 2, 3]
>> h[[1,2]]                   ❹
=> nil
```

❶ Создается хеш (отображение, или ассоциативный массив), отображающий массив в число...
❷ ...который, как предполагается, возвращает число, соответствующее искомому массиву.
❸ Если какой-либо ключ хеша изменится...
❹ ...попытки отыскать массив, который прежде обнаруживался без проблем, будут терпеть неудачу[1].

Проблемы, подобные этой, присутствуют во всех языках программирования, где имеются изменяемые объекты, но их влияние особенно пагубно в языках, где неизменяемые значения используются редко: многие программисты вынуждены изучать сложные в применении

[1] Да, мы могли бы воспользоваться методом хеша `rehash`, что «решило» бы проблему. Такой способ пригоден, когда имеется возможность перестроить ассоциативный массив перед каждой попыткой поиска, но он может снижать эффективность, блокируя доступ к ассоциативному массиву на период перестройки и поиска в хеше, в многопоточной среде выполнения.

приемы решения подобных проблем. Поэтому, даже когда для решения некоторой задачи наиболее эффективным, к примеру, является использование ассоциативного массива с ключами-коллекциями, прошлый опыт мешает им использовать простые решения и вынуждает использовать более сложные и в общем случае менее эффективные подходы. Что лучше? Создать новый класс, содержащий ассоциативный массив и имеющий ограниченный набор операций с ним? Использовать простые неизменяемые коллекции, обеспечивающие адекватную семантику, но требующие выполнять полное копирование, чтобы создать измененную версию? Как оказывается, выбор не прост.

В противоположность этому, в языке Clojure можно безопасно использовать коллекции в ассоциативных массивах (а также в множествах, векторах или записях), не заботясь о значениях ключей или об особенностях использования в многопоточной среде, потому что структуры данных в Clojure являются неизменяемыми и эффективными:

```
(def h {[1 2] 3})                ❶
;= #'user/h
(h [1 2])
;= 3
(conj (first (keys h)) 3)        ❷
;= [1 2 3]
(h [1 2])                        ❸
;= 3
h                                ❹
;= {[1 2] 3}
```

❶ Создается ассоциативный массив Clojure, отображающий ключ-вектор в числовое значение и обеспечивающий ожидаемую семантику поиска.

❷ мы можем «добавить» значение в ключ-вектор и получить в результате «измененный» вектор. Но...

❸ ...это никак не повлияет на сам ассоциативный массив, на вектор, используемый в качестве ключа, и на успешные в прошлом попытки выполнить поиск. Все это обусловлено тем, что...

❹ ...ключ-вектор не изменился и в действительности *никогда не изменится*. При добавлении нового значения в действительности был создан совершенно новый вектор.

Но это лишь вершина айсберга. Понимание и умение использовать абстракции и реализации структур данных в языке Clojure является залогом эффективного владения языком. Пока что, в этой главе,

мы будем использовать структуры данных Clojure неформально, но всю следующую главу мы посвятим их детальному исследованию.

Важность выбора

В общем случае использование объектов, не ограничивающих возможность их изменения, означает, что[1]:

- изменяемые объекты не могут безопасно передаваться методам;
- изменяемые объекты не обеспечивают должной надежности при использовании в качестве ключей в ассоциативных массивах, элементов в множествах, и так далее, потому что их семантика равенства может изменяться с течением времени;
- изменяемые объекты не обеспечивают должной надежности при кешировании;
- изменяемые объекты не могут безопасно использоваться в многопоточной среде выполнения, потому что требуют синхронизации выполнения операций с ними в разных потоках.

Целые классы ошибок, возникающие при использовании изменяемых объектов, становятся просто невозможными при переходе к использованию неизменяемых значений. Отрицательные последствия изменчивости в действительности изучены достаточно давно[2] и проявляются во всех языках, где отсутствуют простые неизменяемые альтернативы. Эти проблемы настолько общие, что в объектно-ориентированном мире был разработан целый комплекс механизмов копирования для решения проблем, связанных с неограниченной изменяемостью состояния, включая.

Конструкторы копий и методы глубокого копирования, гарантирующие безопасный доступ к объекту в определенном состоянии[3].

---

[1] Брайен Гетц (Brian Goetz) подробно рассматривает все эти проблемы в статье по адресу http://www.ibm.com/developerworks/java/library/j-jtp02183.html.

[2] Например, Джошуа Блох (Joshua Bloch), один из главных идеологов стандартной библиотеки Java, рекомендовал «минимизировать изменчивость» (http://www.artima.com/intv/bloch11.html) и в своей книге «Effective Java» (Д. Блох, «Java. Эффективное программирование», Лори, 2002, ISBN: 5-85582-169-2. – *Прим. перев.*), в вышедшей еще в 2001 году, говорил, что: «Классы должны оставаться неизменяемыми, если нет веской причины делать их изменяемыми».

[3] В некоторых ситуациях дело доходит даже до использования сериализации в качестве механизма глубокого копирования, когда объектная модель не предусматривает наличия надежного конструктора копий или методов дублирования.

Множество шаблонов проектирования, направленных на отслеживание изменений и управления ими, включая шаблоны «Наблюдатель» («Observer»), «Реактор» («Reactor») и другие[1].

Массу утилит, создающих относительно тонкие неизменяемые обертки вокруг изменяемых структур данных, таких как `java.util.Collections`. Такие обертки практически всегда делегируют выполнение операций изменяемым коллекциям, лежащим в их основе. То есть, если сама коллекция изменяется, «неизменяемая» обертка вокруг этой коллекции тоже изменяется.

Кипы документации и вредных советов[2], накопившиеся за долгие годы и описывающие, как адекватно управлять одновременным доступом и изменением совместно используемого состояния посредством блокировок. В результате взаимоблокировки и состояния гонки за ресурсами стали одним из основных источников проблем с качеством во всей индустрии.

Наконец, значительная часть проблем в программировании сопряжена с идентификацией и обеспечением *неизменности* повсюду, где только возможно – в алгоритмах, приложениях или бизнес-правилах, которые не должны изменяться. Чем большей степени неизменности удастся добиться, тем плотнее вы сможете сосредоточиться на локальных эффектах конкретного фрагмента программного кода, и тем большую долю риска вы сможете исключить из системы, созданием которой занимаетесь. Использование неизменяемых значений открывает совершенно новые перспективы – вы получаете абсолютную уверенность, что передача коллекции в функцию не повлечет за собой изменение этой коллекции, что многочисленные потоки выполнения смогут использовать значение без всякого риска нарушить его целостность, причем без потери эффективности за счет использования сложных стратегий блокирования доступа, и что зависящие от времени изменения не повлекут за собой зависимость поведения программного кода от времени[3]. Все это уже гарантиру-

---

[1] В главе 12 будут представлены примеры, демонстрирующие, как возможности языка Clojure делают ненужными многие объектно-ориентированные шаблоны проектирования.

[2] Возможно, вы помните неразбериху и неопределенность, вокруг *блокировок с двойной проверкой* (double-checked locking) несколько лет тому назад, которые в конечном итоге были устранены за счет некоторого усложнения и введения новой модели управления памятью в JVM: http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[3] Гейзенбагов, https://ru.wikipedia.org/wiki/Heisenbug.

ется при работе с такими значениями, как числа, и нет причин не потребовать то же самое от структур данных, примером чему могут служить неизменяемые коллекции и записи в языке Clojure.

Функции, как сущности первого порядка, и функции высшего порядка

Несмотря на различия толкования термина «функциональное программирование» в разных языках, одно из требований остается неизменным: функции сами должны быть значениями, чтобы их можно было интерпретировать как обычные данные, принимать в виде аргументов и возвращать в виде результатов из других функций.

Возможность интерпретировать функции как данные позволяет создавать абстракции, отсутствующие в других языках. В качестве простого примера представьте, что требуется написать функцию, которая дважды вызывает некоторую другую функцию. При этом создаваемая функция `call_twice` должна быть достаточно универсальна, чтобы вызывать не какую-то конкретную, а *любую* функцию с *любым* аргументом.

Эта задача легко решается в языках, где функции являются сущностями первого порядка. В Ruby[1] и Python:

```
# Ruby
def call_twice(x, &f)
     f.call(x)
     f.call(x)
end

call_twice(123) {|x| puts x}

# Python
def call_twice(f, x):
     f(x)
     f(x)

call_twice(print, 123)
```

[1] *Блоки* в языке Ruby, объекты, созданные посредством `lambda` или `Proc.new`, и даже методы классов `SomeClass.method(:foo)` – все это разновидности функций, как сущностей первого порядка. С другой стороны, все эти роли в языке Clojure играют функции.

В языке Clojure реализация функции выглядит ничуть не сложнее:

```
(defn call-twice [f x]
    (f x)
    (f x))

(call-twice println 123)
; 123
; 123
```

Однако в Java написать даже такую простую функцию будет значительно сложнее. По иронии, основной язык виртуальной машины JVM не позволяет интерпретировать функции как сущности первого порядка. В языке Java программный код может существовать только в методах, связанных с классами, и на Java-методы нельзя сослаться, не прибегая к механизму рефлексии.

Классы, содержащие только статические вспомогательные методы, такие как `java.lang.Math`, можно с натяжкой считать неким подобием пространств имен, компенсирующим отсутствие поддержки функций, как сущностей первого порядка. Другие вспомогательные методы – подобно строковым операциям, реализованным в классе `java.lang.String` – не являются статическими и потому должны вызываться относительно конкретного экземпляра.

```
Math.max(a, b);

someString.toLowerCase();
```

Такая архитектура может показаться разумной – что может быть проще, чем группировать родственные методы в выделенных классах или привязывать операции со значениями определенного типа к экземплярам этого типа?

В действительности, не делать ни те ни другие значительно проще и во многих отношениях эффективнее.

Например, представьте, что вам потребовалось на языке Java реализовать поиск наибольшего числа в массиве или преобразовать список строк в аналогичный список, но содержащий только знаки нижнего регистра.

Пример 2.2. Некоторые статические вспомогательные методы на языке Java

```
public static int maxOf (int[] numbers) {
     int max = Integer.MIN_VALUE;
     for (int i : numbers) {
          max = Math.max(i, max);
     }
     return max;
}

public static void toLowerCase (List<String> strings) {
     for(ListIterator<String> iter = strings.listIterator(); iter.hasNext();)
     {
          iter.set(iter.next().toLowerCase()); ❶
     }
}
```

❶ Внимание: этот метод изменяет список строк `strings` на месте... мы надеемся, что в программе больше нет ссылок на этот список, где его изменение оказалось бы неожиданностью.

В языке Clojure, напротив, все функции являются *сущностями первого порядка*. Они могут вызываться непосредственно (без использования имен классов или пространств имен), передаваться в виде аргументов и возвращаться в виде результатов другими функциями. Они являются данными, так же как обычные структуры данных, числа и строки.

В языке Clojure имеются функции `max` и `lower-case` (последняя – в пространстве имен `clojure.string`), соответствующие методам `Math.max` и `String.toLowerCase`, использованным выше[1]. Конечно, они могут использоваться непосредственно:

```
(max 5 6)
;= 6
(require 'clojure.string)
;= nil
(clojure.string/lower-case "Clojure")
;= "clojure"
```

[1] В действительности эти функции в языке Clojure делегируют выполнение операций соответствующим Java-методам.

Но сейчас речь не об этом. Благодаря тому, что функции в языке Clojure являются значениями, они могут использоваться с *функциями высшего порядка* (Higher-Order Functions, HOF или ФВП), то есть, с любыми функциями, принимающими другие функции в качестве аргументов или возвращающими функции в качестве результата.

В состав языка Clojure входит достаточно большое количество функций высшего порядка, чтобы всесторонне обсудить их. В ходе обсуждения мы подробно будем останавливаться на наиболее важных из них, включая `map`, `reduce`, `partial`, `comp`, `complement`, `repeatedly` и других. А сейчас познакомимся поближе с функцией `map`, как одной из наиболее часто используемых функций высшего порядка в Clojure[1].

map. Функция `map` принимает единственный аргумент-функцию и один или более аргументов-коллекций, и возвращает последовательность результатов применения этой функции к членам коллекций. Выражаясь формальным языком, результатом вызова `(map f [a b c])` является последовательность `[(f a) (f b) (f c)]`, результатом вызова `(map f [a b c] [x y z])` является последовательность `[(f a x) (f b y) (f c z)]`, и так далее[2].

Дополнительные примеры позволят получить более ясное представление о семантике функции `map`:

```
(map clojure.string/lower-case ["Java" "Imperative" "Weeping"          ❶
                                "Clojure" "Learning" "Peace"])
;= ("java" "imperative" "weeping" "clojure" "learning" "peace")
(map * [1 2 3 4] [5 6 7 8])                                            ❷
;= (5 12 21 32)
```

❶ Простейший случай применения функции `map`: получает функцию `lower-case` и коллекцию строк, и возвращает последовательность тех же строк, в которых все символы приведены к нижнему регистру.

---

[1] Вероятнее всего это обусловлено высокой практичностью абстракции последовательностей, описываемой в разделе «Последовательности», в главе 3.

[2] Если говорить совсем формально, результатом действия функции `map` является `cons`-ячейка, содержащая первым элементом результат `(f a)`, а вторым – ленивую последовательность, представляющую остальные вычисления. Именно из-за «ленивости» последовательности, в действительности возвращается список, а не вектор. Таким образом, правильный результат – `'((f a) (f b) (f c))`. – *Прим. ред.*

❷ Получает * и n коллекций чисел, и возвращает последовательность произведений соответствующих чисел из каждой коллекции.

Первый пример выше – это вариант реализации на языке Clojure статического метода toLowerCase из примера 2.2. Как видите, контраст между двумя подходами оказывается достаточно разительным, даже в таком простом случае.

- Статический метод toLowerCase изменяет свой аргумент на месте, тогда как функция map, подобно другим добропорядочным функциям в языке Clojure, возвращает неизменяемые значения.
- Если бы мы решили переписать метод toLowerCase так, чтобы он возвращал новую коллекцию, содержащую преобразованные строки, нам пришлось бы явно определить тип возвращаемой коллекции и разместить ее в памяти. Функция map всегда возвращает последовательность[1].
- В Java нам постоянно приходится контролировать порядок выполнения, вручную реализуя итерации по входной коллекции и выполняя обход элементов коллекций в определенном порядке. Генераторы списков (list comprehensions) в Python и идиома each в Ruby обеспечивают превосходство над Java в этом отношении. Но язык Clojure пошел еще дальше, позволяя отделять операции от точки их применения. Например, операция «отображения» (mapping), применяемая к последовательности или множеству последовательностей, не гарантирует применение указанной функции к коллекциям в каком-то определенном порядке или даже в одном и том же потоке выполнения[2].

---

[1] Последовательности (sequences), возвращаемые функцией map, являются *ленивыми* (lazy), как и последовательности, возвращаемые многими другими функциями в языке Clojure. Подробнее о ленивых последовательностях рассказывается в разделе «Ленивые последовательности», в главе 3, а пока можно просто игнорировать эту особенность.

[2] Это чрезвычайно удобно и обеспечивает возможность многопоточной обработки данных, как это делает функция pmap, которую можно использовать для параллельного применения *чистых* (pure) функций к коллекции (или коллекциям). Более подробно об чистых функциях рассказывается в разделе «Чистые функции», ниже, а дополнительные сведения о функции pmap и родственных ей механизмах параллельной обработки вы найдете в разделе «Параллельная обработка по невысокой цене», в главе 4.

Функция `map` является фундаментальной функцией высшего порядка для преобразования содержимого любых упорядоченных коллекций, однако часто бывает необходимо на основе коллекции получить единственное значение, не являющееся последовательностью. Для таких случаев предназначена функция `reduce`.

reduce. Создание некоторого значения за счет применения функции к коллекции во многих кругах называется *сверткой* (reduction). В языке Clojure это понятие реализовано в виде функции высшего порядка с именем `reduce`[1]. Простейшим примером применения `reduce` может служить реализация на языке Clojure статического метода `maxOf` из примера 2.2:

```
(reduce max [0 -3 10 48])
;= 48
```

Получив функцию и коллекцию для обработки, `reduce` применяет функцию к каждому элементу коллекции, аккумулирует полученные результаты и возвращает единственное (последнее) значение как результат. Чтобы понять, как действует функция `reduce`, необходимо разобраться с тем, как она применяет переданную ей функцию к элементам коллекции. Если бы потребовалось выполнить те же операции вручную, это выглядело бы так:

```
(max 0 -3)
;= 0
(max 0 10)
;= 10
(max 10 48)
;= 48
```

или, если представить в виде единственного выражения[2]:

```
(max (max (max 0 -3) 10) 48)
;= 48
```

---

[1] И с довольно странным именем `inject` в языке Ruby.

[2] В языке Clojure имеется масса возможностей, позволяющих избежать такого кода. Например, функция `max` может принимать переменное число аргументов, поэтому выражение `(max 0 -3 10 48)` выглядит намного предпочтительнее в сравнении с бессмысленным вложением круглых скобок, которые показаны здесь исключительно в иллюстративных целях.

В первой «итерации» функция `reduce` применяет указанную функцию к первым двум элементам коллекции. Затем `reduce` применяет функцию к предыдущему результату (значение `0` в примере выше) и следующему элементу коллекции (значение `10` в примере выше, так как перед этим функция была применена к элементам `0` и `-3`), чтобы получить следующий промежуточный результат, и так далее. При необходимости функции `reduce` можно передать начальное значение:

```
(reduce + 50 [1 2 3 4])
;= 60
```

Данный пример использования `reduce` не отличается чем-то особенным, кроме того, что при первом вызове функции, указанной в первом аргументе, ей передается начальное значение, а во втором – первый элемент коллекции. Возможность передачи начального значения является особенно важной, так как в результате свертки коллекции значений позволяет легко получить значение любого типа. Например, коллекцию чисел можно превратить в ассоциативный массив с числами в роли ключей и их квадратами в роли значений:

```
(reduce
    (fn [m v]                    ❶
        (assoc m v (* v v)))
    {}                           ❷
    [1 2 3 4])
;= {4 16, 3 9, 2 4, 1 1}
```

❶ В предыдущих примерах функции `reduce` передавались предопределенные функции, но в данном случае передается функция, определяемая на месте. Она принимает два аргумента, ассоциативный массив (результат, полученный на предыдущем этапе свертки или начальное значение, указанное нами) и следующий элемент коллекции. Определяемая нами функция просто применяет функцию `assoc`[1], добавляя в ассоциативный массив следующий элемент коллекции и его квадрат; `assoc` возвращает новый ассоциативный массив, содержащий элементы ассоциативного массива из первого аргумента, который затем будет использоваться в следующей итерации свертки.

[1] Добавляет новый элемент в ассоциативный массив. Является эквивалентом метода `java.util.Map.put(Object, Object)` в Java или `hash_map[key] = value` в Python и Ruby. Подробнее о функции `assoc`, ассоциативных массивах и других коллекциях в языке Clojure будет рассказываться в главе 3.

❷ Непосредственно перед коллекцией, подвергающейся свертке, указывается «начальное» значение. {} – это литерал пустого ассоциативного массива в языке Clojure.

Когда использовать анонимные функции или литералы функций

В предыдущем примере используется анонимная функция, созданная с помощью формы `fn`. Прием использования анонимных функций, как в данном примере, вместе с `map`, `reduce` и другими функциями высшего порядка, весьма распространен в языке Clojure. Однако, не забывайте также о литералах функций. Как было показано в разделе «Литералы функций» в предыдущей главе, они позволяют избавиться от символа `fn` и явного вектора аргументов, обеспечивая более лаконичную форму записи простых функций.

Например, ниже приводится пример использования `reduce`, переписанный с применением литерала функции вместо более длинной формы `fn`:

```
(reduce
    #(assoc % %2 (* %2 %2))
    {}
    [1 2 3 4])
;= {4 16, 3 9, 2 4, 1 1}
```

Использовать ли более длинную форму определения анонимной функции или литерал функции, в значительной степени является вопросом личных предпочтений. Литералы функций обеспечивают более лаконичную форму записи и их предпочтительнее использовать для определения очень коротких функций. Анонимные функции несут больше информации об аргументах и их назначении. В любом случае, на практике используются обе формы, поэтому вы должны быть знакомы с ними одинаково хорошо.

Между использованием `reduce` с функцией `max` и статическим методом `maxOf` на языке Java, реализованном нами в примере 2.2, существуют те же отличия, что и между использованием `map` с функцией `lower-case` и статическим методом `toLowerCase`: сами операции не зависят от режимов их применения, отсутствует явное управление потоком выполнения, и так далее. Однако самое важное отличие

состоит в том, что в языке Clojure никому и в голову не пришло бы определять функции, аналогичные статическим методам `maxOf` и `toLowerCase` – намного более разумно определить базовые операции, такие как `max` и `lower-case`, и затем применять их к данным с использованием наиболее подходящих функций высшего порядка.

Частичное применение

Применение функции (function application) – это вызов функции с последовательностью аргументов, в противоположность вызовам функций, построенным в соответствии с синтаксическими соглашениями. Например, в Ruby и Python (примеры 2.3 и 2.4) функции могут применяться к массивам или спискам аргументов за счет добавления оператора звездочки перед ссылкой на аргумент:

Пример 2.3. Применение функции (метода) в Ruby

```
>> interval = [-10, 10]
=> [-10, 10]
>> Range.new(*interval)
=> -10..10

>> h = {}
=> {}
>> pair = ['a', 5]
=> ["a", 5]
>> h.store(*pair)
=> 5
>> h
=> {"a"=>5}
```

Это – фундаментальная особенность любого языка программирования, поддерживающего различные идиомы функционального программирования, особенно в случаях, когда заранее неизвестно, какую функцию потребуется вызвать (возможно, переданную в виде аргумента в определенном контексте) со списком аргументов неизвестного размера. Иначе было бы просто невозможно (или пришлось бы реализовать сложный анализ, что увеличивает вероятность ошибок) выделить аргументы из данных и упорядочить их, чтобы сформировать «обычный» вызов функции.

В языке Clojure применение функции выполняется с помощью `apply`:

```
(apply hash-map [:a 5 :b 6])
;= {:a 5, :b 6}
```

Функция `apply` обладает весьма удобной особенностью, позволяя предварять последовательность аргументов любым количеством явных аргументов. Во многих случаях, когда наряду с последовательностью аргументов требуется передать некоторые дискретные значения, эта особенность позволяет избежать необходимости создания новой последовательности:

```
(def args [2 -2 10])
;= #'user/args
(apply * 0.5 3 args)
;= -60.0
```

Итак, применение функции – это вызов функции с передачей ей упорядоченной коллекции аргументов. Функция `apply` должна передать применяемой функции *все* аргументы. Под *частичным применением* подразумевается возможность передачи лишь *некоторой* части аргументов и получения новой функции, которую позднее можно вызывать с остальными, недостающими аргументами, необходимыми оригинальной функции.

Пример 2.4. Частичное применение в Python

```
>> from functools import partial
>> only_strings = partial(filter, lambda x: isinstance(x, basestring))
>> only_strings(['a', 5, 'b', 6])
['a', 'b']
```

Разберемся с тем, что здесь происходит. Функция `partial` принимает некоторую функцию (в данном примере – функция `filter`, но это может быть любая другая функция `f`), а также один или более аргументов для этой функции (в данном примере – предикат, проверяющий принадлежность к строковому типу, но это могут быть любые другие аргументы `a`, `b`, ...), и возвращает новую функцию `g`, хранящую эти аргументы наряду со ссылкой на функцию `f`, как показано на рис. 2.2.

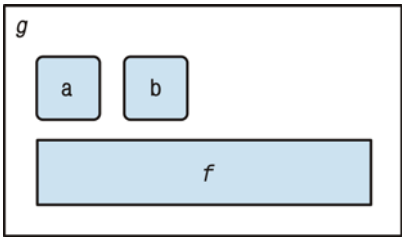

Рис. 2.2. Функция `g`, хранящая переданные аргументы и ссылку на оригинальную функцию

Если вызвать функцию `g`, полученную в результате вызова функции `partial`, она вернет результат вызова оригинальной функции *`f`* с аргументами, переданными функции `partial` (`a, b, ...`), и *дополнительными* аргументами, переданными в вызов функции `g`, как показано на рис. 2.3.

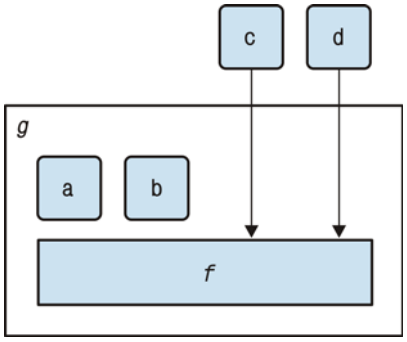

Рис. 2.3. Результат вызова функции `g` с дополнительными аргументами

Частичное применение в языке Clojure обеспечивает функция `partial`:

```
(def only-strings (partial filter string?))
;= #'user/only-strings
(only-strings ["a" 5 "b" 6])
;= ("a" "b")
```

Частичное применение используется в самых разных контекстах. Например, функции, требующие дополнительной настройки (воз-

можно, получающие ссылку на соединение с базой данных или файл для записи), часто получают эти настройки в первых аргументах. Это позволяет с помощью приема частичного применения создавать производные подобных функций, «запирающие» в себе необходимые настройки и вызывать их только с дополнительными аргументами:

```
(def database-lookup (partial get-data "jdbc:mysql://..."))
```

Эта особенность частичного применения – возможность быстро создать производную функцию, не беспокоясь об остальных аргументах, – делает функцию `partial` весьма полезной и привлекательной, особенно в соединении с функцией `comp`[1].

> **Внимание.** Помните, что в сравнении с обычными функциями, имеющими аргументы по умолчанию, функции высшего порядка, такие как `apply` и `partial` дают снижение производительности, хотя и небольшое, и проявляющееся только при большом количестве аргументов: если функции `comp` передается не более трех, а функции `partial` – не более четырех аргументов, используются специальные их реализации, создающие простые замыкания, не требующие упаковки и распаковки списков аргументов переменной длины (varargs).
> Падение производительности функций `apply` и `partial`, связанное с большим числом аргументов, обусловлено необходимостью организовать распаковывание передаваемых последовательностей аргументов, чтобы вызвать функцию с полным списком аргументов. Такое решение не может быть быстрее «обычного» вызова функции, для выполнения которого используется быстрый (действительно быстрый) механизм вызова в JVM. Однако, все не так плохо, потому что благодаря эффективности лежащего в основе механизма, вызовы функций с помощью `apply` и вызовы функций, возвращаемых функцией `partial`, выполняются в языке Clojure быстрее, чем, например, непосредственные вызовы методов в Python и Ruby.

partial в сравнении с литералами функций. Возможно, вы уже обратили внимание, что с технической точки зрения литералы функций являются надмножеством того, что обеспечивает `partial`: они позволяют кратко определять функции, вызывающие другие функции с некоторым подмножеством предопределенных аргументов.

```
(#(filter string? %) ["a" 5 "b" 6])
;= ("a" "b")
```

---

[1] С функцией `comp` мы познакомимся чуть ниже, в разделе «Композиция функций».

Однако литералы функций могут определять не только начальные аргументы в списке:

```
(#(filter % ["a" 5 "b" 6]) string?)
;= ("a" "b")
(#(filter % ["a" 5 "b" 6]) number?)
;= (5 6)
```

Но за это приходится платить необходимостью определять полный список аргументов при вызове функций в них, тогда как `partial` позволяет не заботиться о таких подробностях:

```
(#(map *) [1 2 3] [4 5 6] [7 8 9])                          ❶
;= #<ArityException clojure.lang.ArityException:
;=   Wrong number of args (3) passed to: user$eval812$fn>
 (#(map * % %2 %3) [1 2 3] [4 5 6] [7 8 9])                 ❷
;= (28 80 162)
(#(map * % %2 %3) [1 2 3] [4 5 6])                          ❸
;= #<ArityException clojure.lang.ArityException:
;=   Wrong number of args (2) passed to: user$eval843$fn>
 (#(apply map * %&) [1 2 3] [4 5 6] [7 8 9])                ❹
;= (28 80 162)
(#(apply map * %&) [1 2 3])
;= (1 2 3)

((partial map *) [1 2 3] [4 5 6] [7 8 9])                   ❺
;= (28 80 162)
```

❶ Мы не можем избавиться от необходимости явно определять аргументы функции `map`; так как в результате такого объявления получается литерал функции, принимающей ноль аргументов, потому что количество аргументов определяется наличием ссылок на аргументы внутри литерала.

❷ Эту проблему можно «решить», перечислив аргументы, передаваемые функции `map`...

❸ ...но попытка использовать такой литерал функции без точного соблюдения соответствия с его объявлением потерпит неудачу.

❹ Решением в таких ситуациях является использование функции `apply`, наряду с синтаксической конструкцией `%&`, определяющей в литералах функций список остальных аргументов.

❺ Это практически то же самое, что позволяет функция `partial`, причем гораздо проще и без лишних синтаксических конструкций.

Как видите, в одних ситуациях функция `partial` позволяет получить более удобочитаемый программный код, а в других она дает удобную возможность связать некоторое подмножество начальных аргументов с функцией, общее количество аргументов которой может быть неизвестно заранее.

Композиция функций

Композиция – термин, весьма перегруженный различными смыслами. Мы будем использовать его для обозначения возможности объединения различных частей с целью создать составную функцию, пригодную для многократного использования.

Различные модели программирования предоставляют разные приемы сборки единой конструкции из отдельных частей. Императивный процедурный код обычно вообще не поддается композиции – необходимость вызова функций или методов в ходе выполнения процедуры определяется ее объявлением. Объектно-ориентированная модель предлагает некоторые простейшие средства композиции, наиболее заметным из которых является концепция атрибута, благодаря которой в результате композиции между объектом-владельцем атрибута и другой сущностью устанавливается отношение «имеет» (has-a). Шаблоны проектирования, такие как делегирование (delegation) и подключение (pluggable), являются еще одним способом объединения более мелких частей в единое целое.

Функциональное программирование уделяет возможности композиции еще более пристальное внимание, позволяя начинать с маленьких элементов и объединять их в специализированные абстракции. Дополнительные механизмы создания абстракций в Clojure подробно рассматриваются во второй части книги, однако самой простой абстракцией из всех является функция. Так как они обычно оторваны от данных и в идеале способны полиморфно работать с любыми конкретными типами данных, функции могут без лишних церемоний использоваться для сборки весьма мощных композиций.

Термин *композиция функций* в функциональном программировании имеет совершенно иной смысл[1]: для указанного произвольного

---

[1] В некоторых средах композиция функций оформляется даже специальным образом. Например, в математике композиция функций `f` и `g` обозначается как `f ∘ g`, а в языке Haskell оператор композиции обозначается как точка, `f . g`.

числа функций в результате композиции создается функция, которая передает свои аргументы одной из предоставленных функций, а каждый последующий результат передает в виде аргумента следующей функции, обычно вызывая их в обратном порядке.

Например, допустим, что в программе часто приходится вычислять сумму некоторых исходных чисел, изменять ее знак на противоположный и возвращать результат в виде строки. Для этого можно было бы написать такую функцию:

```
(defn negated-sum-str
     [& numbers]
     (str (- (apply + numbers))))
;= #'user/negated-sum-str
(negated-sum-str 10 12 3.4)
;= "-25.4"
```

Композиция функций в языке Clojure, реализованная в виде функции `comp`, позволяет объединять операции, как показано ниже, наиболее кратким и, обычно, наиболее очевидным способом:

```
(def negated-sum-str (comp str - +))
;= #'user/negated-sum-str
(negated-sum-str 10 12 3.4)
;= "-25.4"
```

Эти два определения `negated-sum-str` функционально эквивалентны и отличаются лишь способом создания.

Что происходит в данном случае? Вызов функции `comp` можно представить как определение конвейера: она принимает произвольное количество функций (`str`, `-` и `+` в данном примере, но это могут быть любые произвольные функции *f*, `g` и `h`) и возвращает новую функцию (помеченную буквой `k` на рис. 2.4), принимающую те же аргументы, что и `h` (так как она вызывается первой в конвейере). Эта новая функция сначала вызывает функцию `h`, резуль-

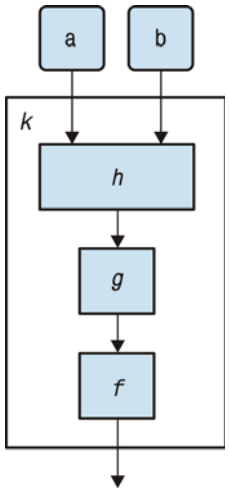

Рис. 2.4. Вызов функции comp можно представить как определение конвейера

тат вызова передает функции g, результат вызова которой передает функции *f*, и так далее.

Результат, полученный вызовом первой функции в списке, возвращается вызывающей программе.

Единственное ограничение функции comp состоит в том, что каждая функция в композиции *должна* возвращать значение, которое может быть передано в виде аргумента синтаксически предшествующей ей функции. То есть, если расположить функции в композиции negated-sum-str в обратном порядке, мы получим ошибку:

```
((comp + - str) 5 10)
;= #<ClassCastException java.lang.ClassCastException:
;= java.lang.String cannot be cast to java.lang.Number>
```

...потому что результат вызова (str 5 10) является строкой и не может служить аргументом операции отрицания (-).

Функция comp может использоваться не только для демонстрации игрушечных примеров, но и для составления весьма сложных цепочек обработки, как из обычных функций, так и из функций, которые сами являются результатом композиции с помощью comp. Например, многие идентификаторы – в программном коде на Java, в разметке на языке XML или других языках, и так далее – часто записываются буквами переменного регистра (в стиле CamelCase). Мы могли бы использовать данные с такими идентификаторами в качестве ключей в ассоциативном массиве Clojure, и для ссылки на эти записи в программном коде Clojure продолжать идиоматически использовать ключевые слова, в которых слова отделяются дефисами и состоят только из букв нижнего регистра.

Реализовать это можно с помощью функции, созданной с применением comp, как показано ниже:

```
(require '[clojure.string :as str])                                    ❶

(def camel->keyword (comp keyword
                          str/join
                          (partial interpose \-)
                          (partial map str/lower-case)
                          #(str/split % #"(?<=[a-z])(?=[A-Z])")))      ❷
;= #'user/camel->keyword
(camel->keyword "CamelCase")
;= :camel-case
(camel->keyword "lowerCamelCase")
;= :lower-camel-case
```

❶ Функция `require` в этом примере обеспечивает загрузку пространства имен `clojure.string` и возможность использования префикса `str` для доступа к его переменным из текущего пространства имен.

❷ Функции `comp` можно передавать не только предопределенные функции. Этот литерал функции выполняет разбиение строк идентификаторов в стиле `CamelCase` между буквами нижнего и верхнего регистров, используя регулярное выражение[1].

Примечание. Того же эффекта можно было бы добиться с помощью макросов `->` и `->>`[2]. Будучи макросами, они не действуют поверх функций, а просто переупорядочивают код в «поток», передавая каждой форме в нем значение или коллекцию в виде первого или последнего аргумента. Например, следующая функция эквивалентна функции `camel->keyword`, созданной выше с помощью `comp`:

```
(defn camel->keyword
     [s]
     (->> (str/split s #"(?<=[a-z])(?=[A-Z])")
          (map str/lower-case)
          (interpose \-)
          str/join
          keyword))
```

Выбор между функцией `comp` или макросами образования потока в значительной степени является вопросом выбранного стиля[3].

Функцию `camel->keyword` можно использовать как часть следующей композиции, определяющей функцию, которая возвращает идиоматический ассоциативный массив Clojure, получая его из последовательности пар ключ/значение, где в качестве ключей используются идентификаторы в стиле `CamelCase`:

```
(def camel-pairs->map (comp (partial apply hash-map)
                            (partial map-indexed (fn [i x]
                                                   (if (odd? i)
```

---

[1] Литералы регулярных выражений поддерживаются механизмом чтения, как было описано в разделе «Регулярные выражения» в главе 1.

[2] Макросы -> и ->> подробно будут обсуждаться в разделе «Подробности: -> и ->>», в главе 5.

[3] Функции `comp` и `partial` поддерживают бессылочный (point-free) стиль (известный также как неявное программирование (tacit programming)), отличительной особенностью которого является возможность определения функций без явного упоминания или обращения к аргументам («ссылкам»).

```
                                                    x
                                                    (camel->keyword x))))))
;= #'user/camel-pairs->map
(camel-pairs->map ["CamelCase" 5 "lowerCamelCase" 3])
;= {:camel-case 5, :lower-camel-case 3}
```

Создание функций высшего порядка

Композиция функций, реализованная в виде `comp`, – это лишь один из множества способов композиции функциональности, безусловно полезный, широко используемый и являющийся в некотором смысле наименьшим общим знаменателем. Несмотря на большое количество универсальных функций высшего порядка в языке Clojure, они совсем необязательно должны быть универсальными.

Более сложные и иногда более практичные композиции возможны только если вы определяете собственные правила взаимодействий между обычными функциями и функциями высшего порядка. Привыкнув к мысли, что функции являются всего лишь одной из разновидностей данных, вы поймете, насколько просто и естественно пишутся функции, принимающие и возвращающие другие функции.

Рассмотрим несколько простейших примеров. В первом примере представлена функция высшего порядка, возвращающая функцию, которая складывает указанное число со своим аргументом:

```
(defn adder
  [n]
  (fn [x] (+ n x)))
;= #'user/adder
((adder 5) 18)
;= 23
```

В следующем, чуть более сложном примере, представлена функция высшего порядка, удваивающая результат, возвращаемый переданной ей функцией:

```
(defn doubler
  [f]
  (fn [& args]
    (* 2 (apply f args))))
;= #'user/doubler
(def double-+ (doubler +))
;= #'user/double-+
```

```
(double-+ 1 2 3)
;= 12
```

Попробуем создать что-то более интересное и попутно познакомимся с некоторыми аспектами функционального программирования, которые в дополнение к операциям с неизменяемыми данными и алгоритмам, действительно очень хорошо подходят для решения проблем, связанных с состоянием и вводом/выводом.

Создание простейшей системы журналирования с применением композиции функций высшего порядка

Журналирование (logging) относится к функциональности, широко востребованной в самых разных приложениях, больших и маленьких, а настройка журналирования часто бывает сложной и громоздкой по целому ряду причин. Реализовать эту функциональность можно несколько иным способом, с помощью пары функций высшего порядка[1]. В процессе мы столкнемся с двумя особенностями языка Clojure, которые пока еще не рассматривали во всех подробностях, но это не должно помешать вам следовать за примерами.

У нас у всех есть дурная привычка использовать `System.out.println`, `puts` или `print` для нужд журналирования – это не очень элегантно, но вполне подходит для крайнего случая. Чтобы улучшить ситуацию, начнем с создания простой функции высшего порядка (ФВП), возвращающей функцию, которая выводит сообщение в указанный нами поток вывода:

```
(defn print-logger
  [writer]                    ❶
  #(binding [*out* writer]    ❷
    (println %)))             ❸
```

❶ ФВП принимает в виде единственного аргумента экземпляр любого класса, наследующего `java.io.Writer`, базовый тип для записи текстовых данных в произвольный поток ввода/вывода.

[1] То, что мы попытаемся реализовать в этом разделе, ни в коей мере не может рассматриваться как замена универсальной библиотеки журналирования `tools.logging`, доступной по адресу: https://github.com/clojure/tools.logging.

❷ `print-logger` возвращает функцию, которая *связывает* поток `*out*`[1] со значением `writer`, экземпляром Writer, переданным функции высшего порядка.

❸ Возвращаемая функция выводит единственный аргумент (текст сообщения, которое требуется сохранить в журнале) с помощью `println`, всегда осуществляющей запись в поток `*out*` (который в данной области видимости мы подменили своим значением `writer`).

Посмотрим, как она действует:

```
(def *out*-logger (print-logger *out*))          ❶
;= #'user/*out*-logger
(*out*-logger "hello")                           ❷
; hello
;= nil
```

❶ Здесь мы передали функции `print-logger` механизм записи `*out*`, поэтому все сообщения, передаваемые возвращаемой функции, будут выводиться в поток `*out*`, точнее в поток, с которым было связано значение `*out*` на момент определения функции `*out*-logger`.

❷ `print-logger` всегда возвращает функцию, принимающую единственный аргумент, и эта функция вызывается здесь со строковым аргументом.

В этом примере мы всего лишь реализовали более сложный аналог функции `println`, выводящий текст в `stdout`. Гораздо интереснее выглядела бы реализация журналирования сообщений в буфер оперативной памяти. Такую возможность обеспечивает уже имеющийся класс `java.io.StringWriter` – это реализация абстрактного класса `java.io.Writer`, осуществляющая вывод не в некоторое устройство, а в символьный буфер:

```
(def writer (java.io.StringWriter.))             ❶
;= #'user/writer
(def retained-logger (print-logger writer))
;= #'user/retained-logger
(retained-logger "hello")                        ❷
;= nil
(str writer)                                     ❸
;= "hello\n"
```

[1] По умолчанию поток `*out*` связан с экземпляром `Writer`, осуществляющим вывод в `stdout`. Выполняя повторную его привязку, мы обеспечиваем передачу выводимой информации указанному механизму записи. Подробнее о *связывании* рассказывается в разделе «Динамическая область видимости» в главе 4.

❶ Мы создали экземпляр StringWriter и сохранили его отдельно, чтобы после вызова print-logger для получения функции журналирования в буфер, можно было обращаться к нему.
❷ Вызов полученной функции журналирования ничего не выводит в stdout...
❸ ...потому что println вывела текст в наш экземпляр StringWriter.

Такая реализация выглядит интереснее, но основную ценность любого механизма журналирования представляет возможность вывода сообщений в файлы. Реализация print-logger позволяет легко получить такую возможность: достаточно передать ей произвольную реализацию интерфейса Writer и она вернет функцию, осуществляющую вывод сообщений в этот экземпляр Writer. То есть, если мы сможем получить экземпляр Writer, осуществляющий вывод в файл, желаемая цель будет достигнута.

```
(require 'clojure.java.io)

(defn file-logger
    [file]                                                          ❶
    #(with-open [f (clojure.java.io/writer file :append true)]      ❷
        ((print-logger f) %)))                                      ❸
```

❶ Функция высшего порядка file-logger принимает единственный аргумент file, определяющий файл, куда будут выводиться сообщения. Из-за особенностей семантики clojure.java.io/writer этот аргумент может быть строкой с именем файла, либо экземпляром java.io.File, java.net.URL или java.net.URI.
❷ Литерал функции, возвращаемый функцией file-logger, создает новый экземпляр Writer для вывода в файл, открывает его для записи в конец (благодаря чему исключается вероятность затирания сообщений, записанных ранее) и связывает его с локальным именем f[1].
❸ Вместо повторного связывания *out* и использования println, мы просто вызываем print-logger с нашим экземпляром Writer, представляющим файл, и затем используем полученную в результате функцию для вывода сообщения. Не забывайте, что функции являются обычными значениями и нет никакой необходимости связывать их с переменными верхнего уровня для их вызова – в этом примере функция,

[1] Форма with-open гарантирует закрытие f в конце своего тела. Она является аналогом идиомы try-with-resorce в Java 7 и инструкции with в языке Python. Дополнительные сведения о форме with-open приводятся в разделе «with-open, аналог finally», в главе 9.

возвращаемая функцией `print-logger`, создается, вызывается и затем немедленно уничтожается.

Посмотрим, что у нас получилось:

```
(def log->file (file-logger "messages.log"))
;= #'user/log->file
(log->file "hello")
;= nil

% more messages.log
hello
```

Отлично: мы смогли создать функцию, реализующую вывод сообщений в указанный нами файл (`messages.log` в текущем каталоге) и, в соответствии с нашими ожиданиями, сообщения действительно записываются в файл. Не так плохо для 10 строк кода, и вы легко могли бы добавить комплект дополнительных функций высшего порядка `*-logger`, возвращающих функции, которые осуществляют вывод сообщений в базы данных, очереди сообщений и в другие хранилища, и использовать их взаимозаменяемо по своему желанию.

Но, как быть, если потребуется обеспечить вывод в несколько разных мест? Это не та особенность, о которой должна заботиться функция журналирования. Для этого нужно определить еще одну функцию высшего порядка, возвращающую функцию, которая сама ничего не выводит, но реализует передачу сообщения нескольким функциям журналирования:

```
(defn multi-logger
    [& logger-fns]           ❶
    #(doseq [f logger-fns]   ❷
        (f %)))
```

❶ `multi-logger` принимает произвольное количество функций журналирования.

❷ Функция, возвращаемая ею, выполняет цикл по функциям журналирования в последовательности[1], вызывая каждую из них с сообщением, которое требуется вывести.

[1] Подробнее о форме `doseq` и итерациях через последовательности в языке Clojure рассказывается в разделе «Последовательности не являются итераторами», в главе 3.

Наличие такой функции упрощает определение функций журналирования, производящих вывод сообщений сразу в несколько мест:

```
(def log (multi-logger                                    ❶
          (print-logger *out*)
          (file-logger "messages.log")))
;= #'user/log
(log "hello again")
; hello again
;= nil

% more messages.log
hello
hello again
```

❶ Мы создали новую функцию журналирования, которая, благодаря `multi-logger`, отправляет сообщения в `*out*` и в файл *messages.log*.

Итак, теперь у нас есть возможность осуществлять журналирование сразу в несколько мест.

А теперь рассмотрим еще одно, заключительное усовершенствование нашей маленькой библиотеки журналирования. Так как все функции журналирования, создаваемые нашими функциями высшего порядка, принимают данные одного и того же вида (простую строку, на данный момент), мы могли бы определить другие функции высшего порядка, дополняющие и трансформирующие данные для журналирования разными способами. Журналируемые сообщения практически всегда включают информацию о текущем времени, поэтому для начала можно было бы определить ФВП, добавляющую текущее время в начало сообщения:

Пример 2.5. Добавление «промежуточной функции», включающей текущее время в каждое журналируемое сообщение

```
(defn timestamped-logger
     [logger]
     #(logger (format "[%1$tY-%1$tm-%1$te %1$tH:%1$tM:%1$tS] %2$s"
                      (java.util.Date.) %)))                    ❶

(def log-timestamped (timestamped-logger
                      (multi-logger
                        (print-logger *out*)
                        (file-logger "messages.log"))))
```

```
(log-timestamped "goodbye, now")
; [2011-11-30 08:54:00] goodbye, now
;= nil

% more messages.log
hello
hello again
[2011-11-30 08:54:00] goodbye, now
```

❶ `timestamped-logger` возвращает функцию, которая просто добавляет текущее время в начало строки с сообщением и вызывает функцию журналирования, созданную с помощью ФВП. `format` – это функция из стандартной библиотеки Clojure, вызывающая Java-метод `String.format`, реализующий форматирование различных объектов в стиле `sprintf`.

Трансформировать журналируемые сообщения можно разными способами – добавлять текущее пространство имен, неявную контекстную информацию (например, имя хоста, где выполняется приложение), возможно, номер строки в исходных текстах, откуда произведен вызов функции журналирования[1], и так далее. Что более важно, при встраивании функций журналирования в нетривиальные приложения одним из наиболее существенных усовершенствований была бы возможность использовать более гибкий тип данных, чем строки, и чтобы сам механизм журналирования оставался открытым и легко расширяемым. Например, если бы каждое журналируемое сообщение представляло собой ассоциативный массив[2], его легко можно было бы использовать для журналирования любой информации без усложнения передачи и обработки журналируемой информации в будущем. Такой структурированный подход упростил бы также возможность фильтрации сообщений на основе их «уровня» или важности.

Чистые функции

Использование неизменяемых значений позволяет ликвидировать целые классы ошибок, которые могут возникать при работе

---

[1] Для этого потребуется использовать макрос, возвращающий номер строки из метаданных `:line`. Подробности см. в разделе «Вывод сообщений об ошибках в макросах» в главе 5.

[2] Безусловно, поддержка строковых сообщений должна остаться – совсем несложно организовать вывод сообщения `"foo"`, например, через неявное преобразование в ассоциативный массив вида `{:message "foo"}`.

с данными в программах, однако остается еще множество других видов ошибок, тесно связанных с используемыми подходами к созданию функций, обрабатывающих такие значения. Большинство ошибок обусловлено *побочными эффектами*, то есть изменениями, которые производят функции в своем окружении, помимо возврата результата.

Вернемся к диаграмме на рис. 2.1. Побочные эффекты – это любые изменения, производимые функцией в окружающем ее мире. Любая функция, использующая случайные числа[1], является отличным примером функции с побочным эффектом, так как она:

1. Зависит от состояния генератора случайных чисел[2].
2. Обязательно изменяет состояние этого генератора случайных чисел, вследствие чего после вызова функции, вызывающая программа будет работать уже с иной последовательностью случайных чисел.

> **Примечание.** Любые операции ввода/вывода или изменения любых совместно используемых объектов по определению производят побочные эффекты.

Использование случайных чисел выглядит слишком прозаично, поэтому для наглядности можно привести другие, не самые гипотетические функции[3]:

```
(defn perform-bank-transfer!
  [from-account to-account amount]
  ...)

(defn authorize-medical-treatment!
  [patient-id treatment-id]
  ...)

(defn launch-missiles!
  [munition-type target-coordinates]
  ...)
```

---

[1] Возвращаемыми функциями `rand` и `rand-int` в Clojure.

[2] В случае программ на языке Clojure – от экземпляра `java.math.Random`.

[3] Обычно функциям, изменяющим внешнее состояние, принято давать имена, оканчивающиеся восклицательным знаком, который служит напоминанием, что данная операция имеет побочные эффекты.

Для большей конкретики, рассмотрим простейшую функцию, принимающую имена пользователей Twitter и возвращающую количество их последователей:

```
(require 'clojure.xml)

(defn twitter-followers
  [username]
  (->> (str "https://api.twitter.com/1/users/show.xml?screen_name="
    username)
    clojure.xml/parse
    :content
    (filter (comp #{:followers_count} :tag))
    first
    :content
    first
    Integer/parseInt))

(twitter-followers "ClojureBook")
;= 106
(twitter-followers "ClojureBook")  ❶
;= 107
```

❶ Многократный вызов `twitter-followers` с одним и тем же аргументом *может* давать разные результаты.

Совершенно очевидно, что невозможно детерминированно протестировать функцию, зависящую от состояния генератора случайных чисел. Во всех отношениях то же самое верно для любой функции, зависящей или изменяющей внешнее состояние, так как зачастую очень сложно перечислить (не говоря уже о том, чтобы осмысленно рассуждать о них) все граничные случаи и условия отказа, связанные с этим состоянием. Для этой цели в тестировании используются фиктивные объекты, моделирующие внешние источники или приемники данных с предсказуемым поведением, позволяющие получить желаемые результаты от тестируемой функции. К сожалению, никакой фиктивный объект (а в действительности, никакой тест) никогда не сможет покрыть полный диапазон ситуаций, с которыми может столкнуться программный код в условиях эксплуатации.

В отличие от функций с побочными эффектами, *чистые функции* (pure functions) не зависят от внешних источников данных, не производят побочных эффектов во внешнем окружении и при вызове

с определенным набором аргументов всегда возвращают одно и то же значение[1]. Все арифметические функции являются чистыми. Например, + не зависит от своего окружения, не производит побочных эффектов и всегда возвращает одно и то же значение для данного набора аргументов. То же относится к любой другой операции с неизменяемыми значениями.

В чем преимущество чистых функций?

Мы уже приводили пару практических следствий от использования и создания чистых функций. По ряду причин они могут существенно упрощать разработку программного обеспечения, особенно в сочетании с широким выбором неизменяемых типов данных.

Чистые функции легче поддаются анализу. Вспомните дискуссию в разделе «Сравнение значений изменяемых объектов» о том, как неизменяемые значения позволяют идентифицировать целые новые категории инвариантов. Чистые функции оказывают аналогичную помощь, но еще и в отношении *операций*, а не только данных. Если известно, что некоторая функция `f` всегда возвращает γ, когда вызывается с аргументами α и β, и что она не изменяет базы данных, файлы и не производит чтение из сокетов, тогда вы можете быть уверены, что вызвав `f` в *любом* контексте с теми же аргументами, вы получите те же результаты.

Чистые функции проще тестировать. Это утверждение является естественным следствием предыдущего. Если известно, что функция не имеет побочных эффектов и ее результат однозначно соответствует аргументам, тестирование такой функции существенно упрощается. Вы можете точно определить круг входных значений для каждой функции и так же точно определить диапазон значений, возвращаемых каждой функцией. Поскольку результат чистой функции определяется исключительно входными аргументами, создание фиктивных объектов для тестирования становится ненужным. Эти характеристики позволяют (если возникнет такое желание) все-

---

[1] Эта последняя характеристика называется *идемпотентностью* (idempotence) и является близкой родственницей чистоты функций. Например, функция, всегда возвращающая одно и то же значение для данного набора аргументов, но производящая побочные эффекты (такие как вывод сообщений в файл журнала) является идемпотентной, но не является чистой.

сторонне протестировать чистую функцию до такой степени, которая невозможна в случае с функциями, обладающими побочными эффектами.

Возможно, вы уже действуете подобным образом, помещая максимально возможный объем функциональности в методы, легко поддающиеся модульному тестированию, оставляя проверку функций (намного более сложных для тестирования), влияющих на состояние, за функциональными и интеграционными тестами. Рассуждения в терминах чистых функций может помочь вам еще четче провести границу между этими двумя областями и способствовать увеличению времени, которое вы будете проводить в более комфортной из них.

Чистые функции поддаются кешированию и легко могут применяться в параллельных вычислениях. О выражениях, состоящих только из чистых функций, говорят, что они являются *ссылочно прозрачными* (referentially transparent); в том смысле, что такие выражения семантически неотличимы от их результатов. Например, все следующие выражения эквивалентны, потому что все функции, используемые в них, являются чистыми:

```
(+ 1 2) (- 10 7) (count [-1 0 1])
```

Каждое из этих выражений можно заменить их результатом, значением 3, и это никак не скажется на выполнении программы, включающей их.

В практическом смысле это означает, что результаты чистых функций могут кешироваться, то есть, результат каждого вызова функции можно сохранить, чтобы в дальнейшем, при обращении к функции с теми же аргументами, сразу же вернуть предыдущий результат, не производя повторные вычисления. Этот прием называется *мемоизацией* (memoization) и используется для решения различных проблем, когда стоимость вычислений результата в некоторой функции оказывается слишком высокой, чтобы производить их многократно для одного и того же набора аргументов. В языке Clojure имеется простая реализация этого приема в виде функции `memoize`; если ей передать некоторую функцию, она вернет *мемоизованную* (memoized) ее версию:

```
(defn prime?                                    ❶
    [n]
    (cond
```

```
          (== 1 n) false
          (== 2 n) true
          (even? n) false
          :else (->> (range 3 (inc (Math/sqrt n)) 2)
                     (filter #(zero? (rem n %)))
                     empty?)))

(time (prime? 1125899906842679))                                    ❷
; "Elapsed time: 2181.014 msecs"
;= true
(let [m-prime? (memoize prime?)]
  (time (m-prime? 1125899906842679))
  (time (m-prime? 1125899906842679)))
; "Elapsed time: 2085.029 msecs"                                    ❸
; "Elapsed time: 0.042 msecs"                                       ❹
;= true
```

❶ Сначала определяется функция, проверяющая – является ли указанное целое число простым (здесь используется упрощенная реализация проверки методом деления).

❷ Для больших простых чисел эта функция будет выполняться достаточно продолжительное время.

❸ После мемоизации та же самая функция с тем же аргументом в первый раз будет выполняться столько же времени. Но...

❹ ...при последующих вызовах с тем же аргументом, результат будет возвращаться немедленно, так как он будет сохранен перед этим функцией, возвращенной функцией `memoize`.

Функции с побочными эффектами не являются ссылочно прозрачными, и потому в общем случае не могут быть надежно мемоизованы. Например, подумайте, что получится, если мемоизовать функцию `rand-int`[1]?

```
(repeatedly 10 (partial rand-int 10))
;= (3 0 2 9 8 8 5 7 3 5)
(repeatedly 10 (partial (memoize rand-int) 10))
;= (4 4 4 4 4 4 4 4 4 4)
```

[1] Обратите внимание, что мемоизация самой функции `rand-int` и результата `(partial rand-int 10)` дает одинаковый эффект. Выяснение причин я оставляю за читателями, в качестве самостоятельного упражнения. Подсказка: подумайте о том, какие аргументы передаются мемоизованной функции в каждом случае.

Правильно, мемоизованное выражение перестанет возвращать случайные числа! Поскольку при мемоизации может не производиться вызов оригинальной функции, никакие побочные эффекты этой функции не будут производиться при возврате мемоизованного результата.

Внимание. Все волшебство, которое творит функция `memoize`, основано на сохранении всех аргументов и возвращаемых значений для всех вызовов функции, которую она создает, то есть эти данные не утилизируются сборщиком мусора. По этой причине, если мемоизованная функция вызывается с большим количеством уникальных наборов аргументов, или если аргументы или возвращаемые значения занимают значительный объем, могут возникать «утечки» памяти, что особенно характерно, когда такие функции сохраняются в переменных верхнего уровня.
Решить подобную проблему можно следующим образом:

1. Ограничить область видимости мемоизованных функций. В частности, не определять их в виде переменных верхнего уровня, то есть, создавать локальные мемоизованные версии функций, вызывающие глобальные функции по мере необходимости.
2. Использовать библиотеку `core.memoize` (https://github.com/clojure/core.memoize), реализующую различные стратегии мемоизации, включая возможность ограничения времени хранения кешированных аргументов и возвращаемых значений на основе различных критериев.

Функциональное программирование в реальном мире

В начале нашего обсуждения функционального программирования приводилась диаграмма, характеризующая все программы (рис. 2.1), и мы согласились, что взаимодействовать с потенциально запутанным, ненадежным внешним состоянием необходимо, чтобы писать действительно полезное ПО. Теперь нам хотелось бы показать вам немного откорректированную перспективу, демонстрирующую, как можно пользоваться преимуществами функционального программирования в повседневной практике, не жертвуя даже малой их толикой.

Независимо от того, какого типа программное обеспечение вы создаете, вы наверняка стараетесь оградить большую часть того, что делает ваше творение уникальным, от хаоса, который будет окружать его после того, как оно окажется в реальном мире. Мы создаем модели нашей предметной области, конструируем абстракции для ключевых операций, определяем базовые алгоритмы, пытаемся

избежать повторяющегося кода, разбиваем реализацию на мелкие модули, легко поддающиеся тестированию, и прилагаем все силы, пытаясь отыскать и определить инварианты, чтобы тем самым оградить подконтрольную нам область от жестокого окружения промышленной эксплуатации.

Совокупность всех этих усилий с функциональным подходом может дать огромные преимущества, о чем рассказывалось на протяжении всей главы. Опираясь на такую надежную платформу, вы будете чувствовать себя более уверенно, при решении задач взаимодействия с внешним миром, зная, что по крайней мере одна сторона моста, соединяющего ваше приложение с окружающей средой, достаточно надежна. Все это можно представить в виде немного подправленной диаграммы, изображенной на рис. 2.5.

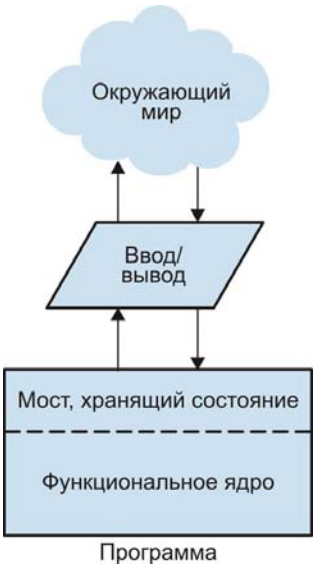

Рис. 2.5. Организация любой программы, написанной в функциональном стиле

Многие привлекательные качества Clojure обусловлены наличием неизменяемых структур данных, функций, как сущностей первого порядка, составных значений и акцентом на уменьшение количества побочных эффектов в целом. Подводя итоги можно сказать, что такая ориентация имеет далеко идущие последствия. Она упрощает анализ программного кода, его тестирование и компоновку, а также делает легкодоступными такие непростые приемы программирования, как организация параллельного выполнения и многозадачность. Именно ФП дает сырье, из которого вам захочется конструировать фундаменты ваших программ.

Глава 3. Коллекции и структуры данных

Основными структурами данных в языке Clojure являются ассоциативные массивы (maps), векторы (vectors), множества (sets) и списки (lists). Как уже было показано выше, каждой из них соответствует своя форма записи литералов:

```
'(a b :name 12.5)         ;; список

['a 'b :name 12.5]        ;; вектор

{:name "Chas" :age 31}    ;; ассоциативный массив

#{1 2 3}                  ;; множество

{Math/PI "~3.14"
 [:composite "key"] 42
 nil "nothing"}           ;; еще один ассоциативный массив

#{{:first-name "chas" :last-name "emerick"}
  {:first-name "brian" :last-name "carper"}
  {:first-name "christophe" :last-name "grand"}} ;;множество ассоц. массивов
```

Эти разновидности структур данных и формы записи их литералов многим покажутся знакомыми. Это особенно относится к языкам Ruby и Python, где аналогичные литералы записываются похожим образом. Однако в языке Clojure структуры данных имеют пару существенных отличий:

1. В первую очередь они используются в терминах абстракций, а не особенностей конкретной реализации.
2. Эти структуры данных являются *неизменяемыми* и *постоянными*, что очень важно с точки зрения эффективности функционального программирования на языке Clojure.

Каждая структура данных обладает собственными характеристиками и идиоматическими шаблонами использования, которые мы будем исследовать в процессе знакомства, но гораздо важнее запомнить отличия, перечисленные выше, и усвоить, что они означают для структур данных в языке Clojure и как с их учетом можно и должно проектировать приложения на Clojure.

Главенство абстракций над реализациями

Лучше иметь 100 функций, которые работают с одной структурой данных, чем 10 функций, работающих с 10 структурами[1].
– Алан Перлис (Alan J. Perlis),
в предисловии к книге «Structure and Interpretation of Computer Programs», http://mitpress.mit.edu/sicp/toc/toc.html[2]

С точки зрения Clojure еще лучше иметь 100 функций для работы с одной *абстракцией*. Главенство абстракций коллекций в Clojure над конкретными реализациями до некоторой степени сродни *полиморфизму* операций в Python и Ruby и использованию интерфейсов в Java, однако имеется ряд тонких (и не очень) отличий, обеспечивающих большую выразительность Clojure и дающих более мощные эффекты.

Чтобы подготовить почву для дальнейшего обсуждения, рассмотрим несколько операций над вектором:

```
(def v [1 2 3])
;= #'user/v
(conj v 4)
;= [1 2 3 4]
(conj v 4 5)
;= [1 2 3 4 5]
(seq v)
;= (1 2 3)
```

[1] Еще больше замечательных «перлизмов» можно найти на сайте http://www.cs.yale.edu/quotes.html.

[2] Харольд Абельсон, Джеральд Джей Сассман, при участии Джулии Сассман, «Структура и интерпретация компьютерных программ», Добросвет, 2006. Доступна по адресу http://newstar.rinet.ru/~goga/sicp/. – *Прим. перев.*

Функция `seq` всегда возвращает представление коллекции в виде *последовательности* (sequence), а функция `conj`[1] добавляет новое значение (или значения) в указанную коллекцию. Вроде бы ничего необычного, однако эта же функция может оперировать ассоциативными массивами:

```
(def m {:a 5 :b 6})
;= #'user/m
(conj m [:c 7])
;= {:a 5, :c 7, :b 6}
(seq m)
;= ([:a 5] [:b 6])
```

...множествами:

```
(def s #{1 2 3})
;= #'user/s
(conj s 10)
;= #{1 2 3 10}
(conj s 3 4)
;= #{1 2 3 4}
(seq s)
;= (1 2 3)
```

...и списками:

```
(def lst '(1 2 3))
;= #'user/lst
(conj lst 0)
;= (0 1 2 3)
(conj lst 0 -1)
;= (-1 0 1 2 3)
(seq lst)
;= (1 2 3)
```

Очевидно, что функции `seq` и `conj` могут оперировать коллекциями любых типов. Функция `conj` добавляет значение в конец вектора или в начало списка, пару ключ/значение – в ассоциативный массив, корректно замещая существующую пару ключ/значение, если таковая имеется, аналогично она добавляет значение в мно-

[1] Имя функции `conj` происходит от «conjoin» – «объединение».

жество, если это значение отсутствует в множестве. Функция `seq` возвращает последовательное интуитивно понятное представление векторов, множеств или списков, а ассоциативные массивы возвращаются в виде последовательностей векторов, состоящих из пар ключ/значение.

Сущность языка Clojure заключается в том, чтобы предоставить простой и удобный прикладной интерфейс (API) для создания вспомогательных функций. С точки зрения пользователя «базовые» и вспомогательные функции ничем не отличаются, и ничто не вынуждает разработчика спешить с созданием новых специализированных вспомогательных функций или с выбором интерфейсов и типов, которые будут поддерживаться данной операцией.

Например, функция `into` действует на основе `seq` и `conj`. Это означает, что она способна оперировать любыми значениями, которые поддерживают функции `seq` и `conj`:

```
(into v [4 5])
;= [1 2 3 4 5]
(into m [[:c 7] [:d 8]])
;= {:a 5, :c 7, :b 6, :d 8}
(into #{1 2} [2 3 4 5 3 3 2])
;= #{1 2 3 4 5}
(into [1] {:a 1 :b 2})          ❶
;= [1 [:a 1] [:b 2]]
```

❶ Для ассоциативных массивов функция `seq` возвращает последовательности векторов, составленных из пар ключ/значение, поэтому при объединении этих пар с некоторым вектором с помощью `conj`, структура пар сохраняется.

В свою очередь, ассоциативные массивы в языке Java даже не являются коллекциями фреймворка `java.util`. Python опирается на конкретные структуры данных, каждая из которых имеет свой набор базовых операций. Положение дел в Ruby несколько лучше, поскольку списки и хеши в нем предоставляют метод `.each` для поддержки императивных итераций, но в остальном это совершенно разные структуры данных с собственными наборами операций. Clojure, в свою очередь, поощряет использование унифицированных абстракций (последовательностей, протоколов, интерфейсов коллекций, и так далее), оставляя вам возможность пойти собственным путем, в зависимости от особенностей поведения конкретных типов.

Дилеммы абстракций

В Java принято разрабатывать простые интерфейсы (частный случай абстракции) и в то же время предоставлять множество вспомогательных функций. Они часто добавляются в интерфейсы (тем самым осложняя их реализацию) и реализуются в абстрактных базовых классах, в которых отсутствуют только реализации основных методов. Из-за чего повторное использование кода становится невозможным без наследования. Язык Java поддерживает только единичное наследование (single inheritance), поэтому, когда возникает необходимость создать новый класс, реализующий пару таких интерфейсов, приходится выбирать, какие вспомогательные методы реализовать повторно, чтобы исключить возможность использования реализаций, существующих в двух абстрактных классах.

Это порождает первую дилемму основных инструментов абстракции в Java и заводит многих разработчиков в ловушку между многоголовым чудовищем и морской пучиной – их собственными Сциллой и Харибдой (http://ru.wikipedia.org/wiki/Сцилла_и_Харибда) – вынуждая делать выбор между необходимостью сопровождать множество реализаций (с низким коэффициентом повторного использования) и небогатым API.

В главе 6 будут представлены некоторые особенности языка Clojure, которые можно использовать при проектировании своих абстракций и типов, придерживаясь при этом принципа главенства обобщенных абстракций.

Создание простых абстракций с широким диапазоном применения является одним из основных принципов проектирования при программировании на языке Clojure, переоценить который невозможно. Легко можно провести аналогию с протоколом HTTP, который обеспечивает гибкость и надежность взаимодействий, определяя всего один простой интерфейс, обычно реализуемый менее чем наполовину. Точно так же многие структуры данных в Clojure поддерживают разнообразные абстракции лишь частично, реализуя интерфейсы коллекций Java, обеспечивающие доступ *только для чтения* (более полные реализации позволили бы изменять коллекции Clojure на месте). Чтобы абстракция имела практическую ценность, не требуется полное соответствие ей всех структур данных, поддерживающих ее[1].

---

[1] В программировании широко применяется хорошо зарекомендовавшая себя стратегия: почти во всех языках имеются различные ошибки и исключения из разряда «операция не поддерживается», используемые для обозначения элементов абстракции, остающихся нереализованными.

Имеется семь основных абстракций, поддерживаемых реализациями структур данных в Clojure:

- коллекция;
- последовательность;
- ассоциативная коллекция;
- индексирование;
- стек;
- множество;
- сортированная коллекция.

Далее мы займемся исследованием семантики операций, определяющих прикладные интерфейсы (API) всех этих абстракций. Попутно мы узнаем, как оперировать структурами данных языка Clojure в терминах этих абстракций, то есть, *любыми* структурами данных, которые поддерживают эти абстракции.

Коллекции

Все структуры данных в языке Clojure поддерживают общую абстракцию *коллекций*. Коллекция – это значение, которое можно использовать в вызовах базовых функций, оперирующих коллекциями:

- `conj` – добавляет элемент в коллекцию;
- `seq` – возвращает коллекцию в виде последовательности;
- `count` – возвращает количество элементов в коллекции;
- `empty` – возвращает пустой экземпляр, тип которого соответствует исходной коллекции;
- `=` – определяет равенство коллекций[1].

Эти функции могут применяться к коллекциям любых типов. Иными словами, семантика каждой операции согласуется с ограничениями реализации каждой структуры данных.

Мы уже встречались с функцией `seq`, и еще будем подробно рассматривать ее в разделе «Последовательности», так как она является ключом к одной из наиболее часто используемых абстракций Clojure – последовательностям.

Точно так же мы видели, как `conj` добавляет элементы в векторы, пары ключ/значение в ассоциативные массивы, и значения в мно-

---

[1] Равенство – весьма тонкий вопрос, касающийся не только коллекций. Более подробно эта тема рассматривается в разделе «Равенство и эквивалентность» в главе 11.

жества (гарантируя членство указанного значения). При этом функция `conj` гарантирует, что операция добавления будет выполняться максимально *эффективно*. Гарантия эффективности имеет неожиданное, на первый взгляд, следствие: из-за особенностей реализации списков (подробнее будут рассматриваться в разделе «Списки»), функция `conj` добавляет элементы в начало списка:

```
(conj '(1 2 3) 4)
;= (4 1 2 3)
(into '(1 2 3) [:a :b :c])
;= (:c :b :a 1 2 3)
```

В противном случае для добавления элемента пришлось бы выполнить обход всего списка, что может оказаться слишком дорогим удовольствием при работе с очень большими наборами данных. Проще говоря, конкретные действия, выполняемые функцией `conj`, зависят от особенностей данной коллекции.

empty. Функция `empty` представляет малознакомую концепцию для большинства. Часто бывает необходимо заранее знать конкретный тип структуры данных, с которой предстоит работать, и поэтому вы просто создаете эту структуру непосредственно. Функция `empty` позволяет уйти от этого шаблона и использовать иной подход, создавая коллекции того же типа, что и имеющиеся экземпляры, например, полученные в виде аргументов. Ниже показано, как можно реализовать перестановку соседних значений в упорядоченной коллекции:

```
(defn swap-pairs
  [sequential]
  (into (empty sequential)
        (interleave
        (take-nth 2 (drop 1 sequential))
        (take-nth 2 sequential))))

(swap-pairs (apply list (range 10)))
;= (8 9 6 7 4 5 2 3 0 1)
(swap-pairs (apply vector (range 10)))
;= [1 0 3 2 5 4 7 6 9 8]
```

Обратите внимание, что `swap-pairs` возвращает значение того же типа, что и ее аргумент: если передать список – она вернет список, если передать вектор – вернет вектор. Такое поведение обеспечи-

вается полиморфизмом функции `into` (подкрепленным функциями `conj` и `seq`) и семантикой функции `empty`, позволяющей получить пустую структуру данных, которая гарантированно будет иметь тот же тип, что и аргумент[1].

Этот прием можно применять не только к упорядоченным последовательностям. Следующая функция позволяет применить указанную функцию к каждому значению в ассоциативном массиве. А как быть, если исходный ассоциативный массив поддерживает сортировку? Никаких проблем, достаточно воспользоваться функцией `empty`, чтобы создать новый экземпляр заданного конкретного типа – в этом случае поддержка сортировки, обеспечиваемая типом исходного ассоциативного массива, сохранится:

```
(defn map-map
  [f m]
  (into (empty m)
        (for [[k v] m]    ❶
          [k (f v)])))
```

❶ `for` – это форма генератора списков, очень близкая по духу генераторам списков в языке Python; она производит «ленивую» (lazy) последовательность векторов пар `[k (f v)]`, по одному для каждой пары ключ/значение `[k v]`, извлекаемых из ассоциативного массива в аргументе `m`.

```
(map-map inc (hash-map :z 5 :c 6 :a 0))
;= {:z 6, :a 1, :c 7}
(map-map inc (sorted-map :z 5 :c 6 :a 0))
;= {:a 1, :c 7, :z 6}
```

Для несортированного ассоциативного массива возвращается несортированный массив, для сортированного – сортированный. Вызывающий программный код получает возможность определять типы возвращаемых значений.

count. Функция `count` делает именно то, что следует из ее имени: подсчитывает количество элементов в коллекции:

---

[1] Сравните эту реализацию, например, со вспомогательными методами коллекций в языке Java, такими как `java.util.List`. Подобные методы могут принимать обобщенные типы (generic types), но для них требуется либо явно указать конкретный тип возвращаемого значения, либо возвращать значение такого же обобщенного типа.

```
(count [1 2 3])
;= 3
(count {:a 1 :b 2 :c 3})
;= 3
(count #{1 2 3})
;= 3
(count '(1 2 3))
;= 3
```

Функция `count` гарантирует эффективную работу с коллекциями любых типов, кроме последовательностей (длина которых может быть не определена, как будет показано ниже).

`count` также с успехом справляется с Java-типами, не являющимися коллекциями Clojure, такими как строки[1], ассоциативные массивы, коллекции и массивы.

Последовательности

Абстракция последовательности (sequence) определяет способ получения и обхода последовательного представления исходных значений, таких как другая коллекция или последовательность результатов некоторых вычислений. Помимо базовых операций, предоставляемых абстракцией «коллекция», последовательности определяют несколько дополнительных операций:

- `seq` преобразует свой аргумент в последовательность;
- `first`, `rest` и `next` позволяют извлекать элементы из последовательностей;
- `lazy-seq` производит «ленивую» последовательность (lazy sequence), элементы которой являются результатом вычисления выражения.

В число типов, пригодных для преобразования в последовательности, то есть типов, для которых функция `seq` сможет создать допустимое значение, входят:

- все типы коллекций в языке Clojure;
- все коллекции в Java (то есть, `java.util.*`);
- все ассоциативные массивы в Java;
- все реализации интерфейса `java.lang.CharSequence`, включая `String`;

[1] В действительности – любыми реализациями интерфейса `java.lang.CharSequence`, включая экземпляры `java.lang.StringBuilder`, `java.lang.StringBuffer` и `java.nio.CharBuffer`.

- любые типы, реализующие интерфейс `java.lang.Iterable`[1];
- массивы;
- `nil` (то есть, `null`, возвращаемый Java-методами);
- любые типы, реализующие интерфейс `clojure.lang.Seqable`.

Полная демонстрация всех возможностей займет слишком много места, поэтому ограничимся лишь некоторыми из них:

```
(seq "Clojure")
;= (\C \l \o \j \u \r \e)
(seq {:a 5 :b 6})
;= ([:a 5] [:b 6])
(seq (java.util.ArrayList. (range 5)))
;= (0 1 2 3 4)
(seq (into-array ["Clojure" "Programming"]))
;= ("Clojure" "Programming")
(seq [])          ❶
;= nil
(seq nil)
;= nil
```

❶ Для значения `nil` или любой пустой коллекции функция `seq` вернет `nil`. Это очень удобно в большинстве случаев, включая проверки условий, такие как `(not (empty? some-collection))`.

Обратите внимание, что многие функции, оперирующие коллекциями, неявно вызывают `seq` для своих аргументов. Например, значение типа `String` необязательно обертывать вызовом `seq`, чтобы передать его функции `map` или `set`:

```
(map str "Clojure")
;= ("C" "l" "o" "j" "u" "r" "e")
(set "Programming")
;= #{\a \g \i \m \n \o \P \r}
```

[1] Функция `seq` оперирует объектами `Iterable` непосредственно; имеются также функции `iterator-seq` и `enumerator-seq`, позволяющие получать последовательности из объектов `java.lang.Iterator` и `java.lang.Enumerator` соответственно. Эти функции отделены от `seq`, потому что являются деструктивными: выполнить обход элементов с помощью интерфейса `Iterator` или `Enumerator` можно *только один раз*, что собственно и происходит при получении последовательностей из них. Функция `seq`, напротив, не является деструктивной в отношении объектов `Iterable`, потому что они позволяют в любой момент получить новый итератор `Iterator`.

Примечание. Функции обретут это поведение автоматически, если будут основаны на использовании других функций, оперирующих последовательностями. Однако, чтобы получить то же самое поведение при использовании `lazy-seq`, необходимо явно применить `seq` к аргументам своей функции.

Обход элементов последовательностей и их обработка могут быть выполнены разными способами, и стандартная библиотека Clojure содержит в пространстве имен `clojure.core` десятки функций для обработки и создания коллекций. Однако наиболее фундаментальными являются `first`, `rest` и `next`:

```
(first "Clojure")
;= \C
(rest "Clojure")
;= (\l \o \j \u \r \e)
(next "Clojure")
;= (\l \o \j \u \r \e)
```

Действие функций `first` и `rest` должно быть очевидным по их именам. С последней из них мы встречались, когда рассматривали особенности определения функций со списком аргументов переменной длины и использовали ее для преобразования «оставшихся аргументов» в последовательность. Можно сказать, что `rest` лежит в основе таких функций[1].

Однако разница между `rest` и `next` не так очевидна: для большинства значений в примерах выше они дают идентичные результаты. Они возвращают разные результаты лишь при обработке последовательностей, содержащих ноль или одно значение:

Пример 3.1. Сравнение rest и next

```
(rest [1])
;= ()
(next [1])
;= nil
(rest nil)
;= ()
(next nil)
;= nil
```

[1] См. раздел «Функции с переменным числом аргументов» в главе 1.

Функция `rest` всегда возвращает пустую последовательность, тогда как `next` возвращает `nil`, если в результате получается пустая последовательность. Проще говоря, следующее выражение всегда будет истинным для любых значений `x`:

```
(= (next x)
   (seq (rest x)))
```

На первый взгляд это отличие кажется несущественным, но оно обеспечивает возможность создания «ленивых» последовательностей. Подробнее о «ленивых» последовательностях и функции `lazy-seq` рассказывается чуть ниже.

Последовательности не являются итераторами

Встретив в программе такой код:

```
(doseq [x (range 3)]
  (println x))
; 0
; 1
; 2
```

можно было бы подумать, что форма `doseq` извлекает значения `x` из некоторого итератора, выполняющего обход упорядоченной коллекции, возвращаемой функцией `range`. Да, действительно переменная `x` последовательно связывается со значениями из последовательности `(range 3)`, но эта последовательность является неизменяемой, как и все остальное в языке Clojure:

```
(let [r (range 3)
      rst (rest r)]
  (prn (map str rst))
  (prn (map #(+ 100 %) r))
  (prn (conj r -1) (conj rst 42)))
; ("1" "2")
; (100 101 102)
; (-1 0 1 2) (42 1 2)
```

«Производная» последовательность `rst` действует точно так же, как «родительская» последовательность `r`, и обе они могут независимо обрабатываться с использованием всего диапазона функций, доступных для коллекций и последовательностей. В частности, к по-

следовательностям можно применять любые функции, что никак не скажется на исходных значениях ни в родительских последовательностях, ни в их «производных». Ни одна из этих особенностей не поддерживается итераторами с изменяемым состоянием и генераторами, которые нельзя зафиксировать в определенном состоянии, не могут служить основой для других итераторов и не способны использоваться многократно.

Последовательности не являются списками

На первый взгляд последовательности очень похожи на списки: они могут быть пустыми или состоять из «головы» (head) и «хвоста» (tail), последний из которых сам является последовательностью. Кроме того, списки сами являются последовательностями[1]. Однако они имеют ряд важных отличий:

- определение длины последовательности является дорогостоящей операцией;
- значения, содержащиеся в последовательностях, могут *вычисляться* только при непосредственном обращении к ним;
- вычисления, производящие значения для «ленивой» последовательности (lazy sequence), могут представлять собой неограниченную прогрессию, производя тем самым *бесконечные* последовательности, длину которых определить невозможно.

Списки, напротив, отслеживают свою длину, поэтому вызов функции `count` является недорогой операцией, с постоянным временем выполнения. Последовательности не могут гарантировать то же самое, потому что они могут быть «ленивыми» и потенциально бесконечными. То есть, единственный способ определить длину последовательности – выполнить обход всех ее элементов. Увидеть последствия этого можно, сравнив время работы функции `count` с «ленивой» последовательностью и с готовым списком:

```
(let [s (range 1e6)]                          ❶
  (time (count s)))
; "Elapsed time: 147.661 msecs"
;= 1000000
(let [s (apply list (range 1e6))]             ❷
  (time (count s)))
; "Elapsed time: 0.03 msecs"
;= 1000000
```

[1] Это – особенность реализации, но в настоящее время выражение `(identical? some-list (seq some-list))` всегда истинно.

❶ range возвращает «ленивую» последовательность чисел, производящую их *только* по требованию. Вызов count – один из способов заставить такую последовательность воспроизвести все свое содержимое, так как для подсчета элементов требуется произвести их все. В этом примере выполняется хронометраж данной операции.

❷ Списки всегда отслеживают свой размер, поэтому вызов функции count для списков практически сразу же возвращает результат..

Создание последовательностей

Возможно, вы уже заметили, что ни в одном примере выше последовательности не создавались явно. В этом есть определенный смысл, учитывая, что последовательности являются всего лишь представлениями других коллекций. В общем случае последовательность может быть создана на основе коллекции либо явно – вызовом функции seq, либо неявно – вызовом другой функции (такой как map), применяющей функцию seq к своим аргументам. Однако существует еще два способа *создания* последовательностей: cons и list*[1].

Функция cons принимает два аргумента – значение, служащее головой новой последовательности, и другую коллекцию, которая после преобразования в последовательность функцией seq будет служить хвостом:

```
(cons 0 (range 1 5))
;= (0 1 2 3 4)
```

Для простоты можно считать, что cons всегда добавляет голову в начало коллекции, представляющей хвост, независимо от ее конкретного типа. Этим она отличается от функции conj:

```
(cons :a [:b :c :d])
;= (:a :b :c :d)
```

Функция list*[2] – это всего лишь вспомогательная функция, позволяющая указать любое количество элементов в голове будущей

---

[1] Если у вас есть опыт использования других диалектов Lisp, обратите внимание, что функция cons в Clojure имеет мало общего с функцией cons в других диалектах. Аналогично списки в языке Clojure не являются последовательностями cons-ячеек.

[2] Может показаться странным, но list* *не* возвращает список, то есть, выражение (list? (list* 0 (range 1 5))) вернет истинное значение, только

последовательности. Таким образом, следующие два выражения эквивалентны:

```
(cons 0 (cons 1 (cons 2 (cons 3 (range 4 10)))))
;= (0 1 2 3 4 5 6 7 8 9)
(list* 0 1 2 3 (range 4 10))
;= (0 1 2 3 4 5 6 7 8 9)
```

Функции `cons` и `list*` обычно используются при создании макросов – где последовательности и списки эквивалентны и требуется добавить значение в начало списка или последовательности – а также при определении следующего шага в «ленивых» последовательностях, о которых рассказывается далее.

Ленивые последовательности

Содержимое последовательностей может вычисляться в отложенном режиме (или «лениво» (lazily)), когда значения элементов производятся по требованию – при попытке обращения к ним. Каждое такое значение вычисляется один и только один раз. Процесс доступа к «ленивой» последовательности называется *реализацией* (realization), а когда все значения «ленивой» последовательности будут вычислены, говорят, что она *полностью реализована* (fully realized).

Создать «ленивую» последовательность можно с помощью макроопределения `lazy-seq`, принимающего любое выражение, возвращающее значение, которое можно преобразовать в последовательность. Например:

```
(lazy-seq [1 2 3])
;= (1 2 3)
```

Это не самый интересный пример, потому что «ленивая» последовательность здесь «реализует» свои значения из полностью сформированной структуры данных. Гораздо интереснее выглядит «ленивая» последовательность случайных целых чисел:

если передать ей единственное значение. Это вызывает проблемы, только когда необходимо использовать структуру данных конкретного типа, чтобы обеспечить определенное поведение программы, используя, например, `list?`. Как обычно, вместо конкретных типов предпочтительнее использовать абстракции, поэтому везде, где может потребоваться `list?`, используйте `seq?` или `sequential?`.

Пример 3.2. Создание ленивой последовательности

```
(defn random-ints
  "Возвращает ленивую последовательность случайных целых чисел в диапазоне
   [0,limit)."
  [limit]
  (lazy-seq
    (cons (rand-int limit)            ❶
          (random-ints limit))))      ❷

(take 10 (random-ints 50))
;= (32 37 8 2 22 41 19 27 34 27)
```

❶ Возвращается «ленивая» последовательность, которая определена головой `(rand-int limit)`...

❶ ... и хвостом, который сам является «ленивой» последовательностью, возвращаемой рекурсивным вызовом самой функции `random-ints`.

Результат вызова не выглядит чем-то особенным, но числа в последовательности, возвращаемой функцией `random-ints`, вычисляются только по требованию – здесь она показана целиком, потому что операция вывода вынуждает последовательность реализовать свое содержимое[1]. Убедимся в этом, немного изменив `random-ints` так, чтобы она выводила каждое реализуемое значение:

```
(defn random-ints
  [limit]
  (lazy-seq
    (println "realizing random number")    ❶
    (cons (rand-int limit)
          (random-ints limit))))

(def rands (take 10 (random-ints 50)))      ❷
;= #'user/rands
(first rands)                               ❸
```

[1] Это означает, что необходимо проявлять осторожность при обращении со ссылками на бесконечные или очень большие последовательности без наложения дополнительных ограничений, как например в выражении `(take 100 infinite-seq)`, извлекающем из последовательности только 100 первых значений. Кроме того, можно установить `set! *print-length*` в некоторое разумное значение (например, 100), чтобы ограничить количество элементов при выводе коллекций.

```
; realizing random number
;= 39
(nth rands 3)
; realizing random number
; realizing random number
; realizing random number
;= 44
(count rands)                                    ❹
; realizing random number
; realizing random number
; realizing random number
; realizing random number
; realizing random number
; realizing random number
;= 10
(count rands)                                    ❺
;= 10
```

❶ Каждый раз, когда выполняется макрос `lazy-seq` (а это происходит, когда производится конкретное значение обращением к голове результата вызова функции `cons`), выводится сообщение.

❷ Определяется «ленивая» последовательность, чтобы избежать операции вывода, которая вызвала бы реализацию всех элементов последовательности. Обратите внимание, что никаких сообщений при этом не выводится – последовательность действительно «ленивая» и к данному моменту не произвела ни одного значения.

❸ Обращение к первому значению в последовательности с помощью `first` *вынуждает* последовательность создать это значение с применением логики, объявленной в форме `lazy-seq`. В результате извлекается случайное число и выводится сообщение.

❹ При обращении к значениям в «ленивой» последовательности, они неизбежно реализуются. Здесь видно, что `count` вынуждает последовательность реализоваться целиком, чтобы определить ее размер.

❺ При повторном вызове `count` (или, например, при обращении к некоторому значению в последовательности с помощью `nth`) элементы последовательности не вычисляются повторно: после реализации значения в ленивой последовательности сохраняются.

Когда «ленивая» последовательность только определяется в переменной, ее содержимое просто не существует (рис. 3.1).

После обращения к первому значению, это значение реализуется и сохраняется, благодаря чему при последующих обращениях к этому значению оно не вычисляется повторно (рис. 3.2).

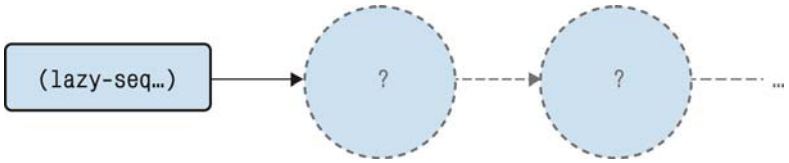

Рис. 3.1. Сразу после определения «ленивой» последовательности ее содержимое не существует

Рис. 3.2. Реализованные значения в последовательности сохраняются

Голова последовательности теперь имеет конкретное значение, но ее хвост полностью определяется отложенными вычислениями в функции, созданной макросом `lazy-seq` из выражения, которая не будет вызываться, пока не произойдет обращение к соответствующему значению. Это – ключевое преимущество функций `cons` и `list*`: они *не* форсируют вычисление (возможно «ленивых») последовательностей, получаемых в последнем аргументе. Эта особенность делает данные функции важным инструментом создания «ленивых» последовательностей при использовании распространенного шаблона, когда `cons` или `list*` применяются для объединения одного или более конкретных значений с результатом вызова макроса `lazy-seq`, который откладывает вычисления оставшейся части последовательности.

Функция `random-ints` в действительности не лучший образец для подражания. Она имеет слишком сложную реализацию функциональности, которую можно получить, воспользовавшись парой функций из стандартной библиотеки Clojure: `rand-int`, уже используемая в `random-ints`, и `repeatedly`, возвращающая «ленивую» последовательность значений, вычисляемых указанной функцией:

```
(repeatedly 10 (partial rand-int 50))
;= (47 19 26 14 18 37 44 13 41 38)
```

Примечание. Важно подчеркнуть, что выражение, передаваемое макросу `lazy-seq`, может выполнять практически любые операции – вы не ограничены операциями с числами, такими как получение случайных чисел, или операциями с небольшими фрагментами данных. В этой игре может участвовать любая чистая функция, реализующая вычисление значений, соответствующих требуемой последовательности.

Обратите внимание, что функция `repeatedly` с единственным аргументом может возвращать бесконечные «ленивые» последовательности случайных чисел. Бесконечные «ленивые» последовательности в языке Clojure совсем не редкость. И в стандартной библиотеке, и в различных библиотеках, создаваемых сообществом Clojure, имеется множество функций, прозрачно работающих с «ленивыми» (и потенциально бесконечными) последовательностями. Например, все основные функции обработки последовательностей в стандартной библиотеке – такие как `map`, `for`, `filter`, `take` и `drop`[1] – возвращают «ленивые» последовательности и могут объединяться в иерархии вызовов, не оказывая влияния на «ленивость» исходных последовательностей. Благодаря этим особенностям, к обработке последовательностей значений можно свести решение многих задач, от таких распространенных, как отложенная обработка данных из очередей и параллельный поиск[2], до отложенного извлечения и преобразования данных из различных источников, например, с использованием `file-seq`, `line-seq` и `xml-seq`.

Иногда бывает желательно сократить до минимума операции, вызывающие реализацию «ленивой» последовательности. Например, когда для получения очередного значения требуется выполнить операции ввода/вывода или произвести массивные вычисления. В таких ситуациях особенно важными становятся различия между `next` и `rest`. Помните, как в примере 3.1 демонстрировалось, что `next` всегда вместо пустой последовательности возвращает `nil`? Это обусловлено тем, что `next` проверяет наличие значений в хвосте последовательности. Данная проверка неизбежно вызывает реализацию головы непустого хвоста последовательности:

---

[1] ...и их производные, такие как `take-nth`, `take-while`, `drop-while`, `remove` и так далее.

[2] Способность фильтровать гигабайты данных в многопроцессорных системах с помощью `pmap` и «ленивых»последовательностей, используя для этого всего пару мегабайтов памяти, может вызвать восторг у кого угодно. Подробнее о `pmap` рассказывается в разделе «Параллельная обработка по невысокой цене», в главе 4.

```
(def x (next (random-ints 50)))
; realizing random number
; realizing random number
```

Функция `rest`, напротив, «слепо» возвращает хвост указанной последовательности, не вызывая реализацию его головы, обеспечивая тем самым дополнительную поддержку «ленивости»:

```
(def x (rest (random-ints 50)))
; realizing random number
```

> **Внимание.** В операциях деструктуризации всегда используется функция `next`, и никогда – `rest`. То есть, деструктуризация «ленивой» последовательности всегда будет вызывать реализацию значения головы ее хвоста:
>
> ```
> (let [[x & rest] (random-ints 50)])
> ; realizing random number
> ; realizing random number
> ;= nil
> ```

С другой стороны, иногда бывает необходимо выполнить полную реализацию «ленивой» последовательности. В таких случаях, если желательно сохранить содержимое последовательности, следует использовать `doall`[1], а если желательно избавиться от значений, производимых последовательностью, следует использовать `dorun`:

```
(dorun (take 5 (random-ints 50)))
; realizing random number
; realizing random number
; realizing random number
; realizing random number
; realizing random number
;= nil
```

Отказ от сохранения содержимого «ленивой» последовательности может показаться пустой тратой вычислительных ресурсов и времени, но иногда такой прием имеет смысл, когда вычисление значений

---

[1] Как было показано выше, `count` (и некоторые другие функции) делают то же самое, но это лишь особенность их реализации. *Возможно* (хотя и редко), что «ленивая» последовательность «знает» свою длину и может сразу вернуть ее как результат `count`, не реализуя свое содержимое.

в «ленивой» последовательности производит побочные эффекты и необходимо обеспечить их выполнение. Например, представьте, что имеется «ленивая» последовательность файлов (полученная с помощью `file-seq`) и необходимо выполнить некоторые операции с файлами, удобно выражающиеся в применении одной или нескольких функций `map` к этой «ленивой» последовательности. В этом случае функция `dorun` позволит применить необходимые операции, не расходуя память на сохранение ненужных значений, возвращаемых функциями, которые вовлечены в операции.

Обычно в документации к функциям, выполняющим операции над последовательностями, явно указывается – производят ли они «ленивые» последовательности или принудительно вызывают их реализацию:

```
(doc iterate)
; -------------------------
; clojure.core/iterate
; ([f x])
;   Возвращает ленивую последовательность x, (f x), (f (f x)) и т.д.
;   f не должна иметь побочных эффектов
(doc reverse)
; -------------------------
; clojure.core/reverse
; ([coll])
;   Возвращает не ленивую последовательность элементов в coll,
;   расположенных в обратном порядке.
```

Очевидно, что `iterate` обеспечивает отложенные вычисления, а `reverse` – нет. Примечание «не должна иметь побочных эффектов» в описании функции `iterate` заслуживает отдельных пояснений.

Код, определяющий «ленивую» последовательность, должен минимизировать побочные эффекты. В разделе «Чистые функции», в главе 2, уже рассказывалось о преимуществах чистых функций, но в контексте «ленивых» последовательностей они (и ловушки, связанные с побочными эффектами) приобретают особое значение. Поскольку значения в ленивых последовательностях реализуются в момент обращения к ним, а не когда они определяются, очень, очень легко потерять контроль над тем, где и когда происходят побочные эффекты, связанные с реализацией этих значений, если они вообще происходят! Например, изменение уровня журналирования в процессе выполнения операций с «ленивой» последовательностью

может вызвать запись в журнал сообщений, которые предусматривалось отфильтровывать на момент определения такой «ленивой» последовательности.

Хуже того, при вычислении некоторых «ленивых» последовательностей может предусматриваться пакетная обработка с целью оптимизации производительности[1], когда выполняется опережающая реализация некоторого количества значений, вызывающая целые серии побочных эффектов, в противоположность последовательному извлечению отдельных значений.

Вывод напрашивается сам собой – не следует полагаться на механизм вычисления последовательностей, как на средство управления потоком выполнения. Механизм отложенных вычислений в Clojure служит совсем иным целям, чем в других языках, где поддерживаются аналогичные механизмы. В Clojure отложенные вычисления возможны только с последовательностями, а остальные структуры данных вычисляются немедленно. «Ленивые» последовательности в Clojure позволяют прозрачно обрабатывать большие объемы данных, не умещающиеся в памяти, и выражать алгоритмы в более однородном, декларативном и конвейерном виде; в этом контексте последовательности можно рассматривать как *своеобразный способ организации вычислений*, а не как коллекции.

Этот шаблон часто можно увидеть в программах на языке Clojure: из одного или нескольких источников данных извлекается последовательность, обрабатывается и преобразуется в более подходящую структуру данных. Этот шаблон проявляется даже в таком простом коде, как показано ниже:

```
(apply str (remove (set "aeiouy")
             "vowels are useless! or maybe not..."))
;= "vwls r slss! r mb nt..."
```

Здесь имеется источник данных, строка `"vowels are useless! or maybe not..."` («гласные не нужны! или нужны...»), который неявно преобразуется в последовательность знаков. Затем из последовательности

[1] Некоторые структуры данных, такие как векторы, генерируют «блочные» последовательности («chunked» sequences), и многие функции обработки последовательностей (например, `map`, `filter`) обрабатывают такие последовательности блоками (по 32 значения, в случае последовательностей, полученных из векторов), а не по одному значению.

удаляются все гласные буквы, и полученная в результате последовательность преобразуется обратно в строку.

Удержание мусора

Начинающие программисты на Clojure часто упускают из виду одно обстоятельство – «ленивые» последовательности не изменяются: элемент, вычисленный однажды, сохраняется в последовательности[1]. Это означает, что пока существует хотя бы одна ссылка на последовательность, ее элементы не могут быть утилизированы сборщиком мусора. Данную проблему называют *удержанием мусора* (head retention). Она может увеличивать нагрузку на виртуальную машину (VM), вызывая снижение производительности и даже ошибки нехватки памяти, если реализованная часть последовательности окажется слишком велика.

`split-with` – это функция, принимающая предикат и значение, которое можно преобразовать в последовательность, и возвращающая вектор с двумя «ленивыми» последовательностями; первая содержит значения, удовлетворяющие предикату, а вторая – не удовлетворяющие:

```
(split-with neg? (range -5 5))
;= [(-5 -4 -3 -2 -1) (0 1 2 3 4)]
```

Рассмотрим случай использования `split-with`, когда известно, что первая последовательность получится очень короткой, а вторая очень длинной. Если удерживать ссылку на первую последовательность, сохранены будет все значения, даже если вторая последовательность будет обрабатываться в отложенном режиме. Если исходная последовательность окажется слишком длинной, тогда проблема удержания мусора (head retention) приведет к ошибке нехватки памяти:

```
(let [[t d] (split-with #(< % 12) (range 1e8))]
  [(count d) (count t)])
;= #<OutOfMemoryError java.lang.OutOfMemoryError: Java heap space>
```

---

[1] Этим они разительно отличаются от генераторов в языке Python и экземпляров `Enumerator` в Ruby, которые не сохраняют ранее вычисленные значения и потому способны воспроизводить бесконечные серии значений по требованию.

Эту проблему можно решить, изменив порядок вычислений:

```
(let [[t d] (split-with #(< % 12) (range 1e8))]
  [(count t) (count d)])
;= [12 99999988]
```

Так как в последнем примере ссылка `t` обрабатывается перед ссылкой `d`, ссылка на голову исходной последовательности, возвращаемой функцией `range`, не сохраняется и переполнения памяти не происходит.

Операция добавления в ассоциативный массив или множество, `=` и `count` также подвержены проблеме «удержания мусора» (head retention)[1] поскольку они вызывают полную реализацию «ленивых» последовательностей.

Ассоциативные коллекции

Абстракция *ассоциативных коллекций* используется структурами данных, связывающих ключи и значения некоторым способом. Она определяется четырьмя операциями:

- `assoc` – устанавливает новые взаимосвязи между ключами и значениями в указанной коллекции;
- `dissoc` – удаляет взаимосвязи с указанными ключами из коллекции;
- `get` – отыскивает значение, соответствующее указанному ключу в коллекции;
- `contains?` – предикат, возвращающий `true`, только если коллекция содержит значение, связанное с указанным ключом.

Канонической ассоциативной структурой данных является ассоциативный массив (map), который, к тому же, является самой универсальной структурой данных в языке Clojure. Вы очень быстро обнаружите, что ассоциативные массивы – ваши лучшие друзья.

Как уже было показано выше, при обработке основными функциями для работы с коллекциями, `conj` и `seq`, ассоциативные массивы рассматриваются как коллекции пар ключ/значение, но более естественно применять к ним ассоциативные функции.

```
(def m {:a 1, :b 2, :c 3})
;= #'user/m
```

[1] Точнее, они вызывают преждевременную реализацию, что, впрочем, также влечет ошибку нехватки памяти.

```
(get m :b)
;= 2
(get m :d)
;= nil
(get m :d "not-found")
;= "not-found"
(assoc m :d 4)
;= {:a 1, :b 2, :c 3, :d 4}
(dissoc m :b)
;= {:a 1, :c 3}
```

Функции `assoc` и `dissoc` также удобно использовать для обработки сразу нескольких элементов в указанном ассоциативном массиве:

```
(assoc m
  :x 4
  :y 5
  :z 6)
;= {:z 6, :y 5, :x 4, :a 1, :c 3, :b 2}
(dissoc m :a :c)
;= {:b 2}
```

Функции `get` и `assoc` чаще используются с ассоциативными массивами, однако они также могут применяться к векторам. На первый взгляд это кажется противоестественным, тем не менее, *обе* структуры, ассоциативные массивы и векторы, являются ассоциативными коллекциями, только в векторах значения связываются с числовыми индексами:

```
(def v [1 2 3])
;= #'user/v
(get v 1)                    ❶
;= 2
(get v 10)
;= nil
(get v 10 "not-found")
;= "not-found"
(assoc v
  1 4
  0 -12
  2 :p)                      ❷
;= [-12 4 :p]
```

❶ вектор v «ассоциирует» индекс 1 со значением 2.
❷ С помощью assoc индексы в векторе можно связать с другими значениями; это не противоречит абстракции, но эффект зависит от типа структуры данных. Обратите внимание, что этот вызов assoc структурно идентичен вызову assoc в предыдущем примере, где устанавливаются взаимосвязи с несколькими ключами в ассоциативном массиве.

Функция conj является наиболее эффективным инструментом добавления значений в вектор. Однако ту же операцию с векторами можно выполнять с помощью assoc, только при этом необходимо заранее знать индекс нового элемента:

```
(assoc v 3 10)
;= [1 2 3 10]
```

Наконец, функция get может работать с множествами, и возвращает «ключ», если он присутствует:

```
(get #{1 2 3} 2)
;= 2
(get #{1 2 3} 4)
;= nil
(get #{1 2 3} 4 "not-found")
;= "not-found"
```

Множества сами по себе не поддерживают семантику пар ключ/значение, однако при работе с множествами функция get считает, что значения в множестве «ассоциированы» сами с собой. Это может показаться странным, но эта особенность позволяет множествам удовлетворять семантике функции get, не вступая в противоречие с типичными приемами их использования, как в следующем примере:

```
(when (get #{1 2 3} 2)
  (println "it contains `2`!"))
; it contains `2`!
```

Примечание. Ассоциативные массивы и множества в Clojure поддерживают семантику get непосредственно и могут использоваться без функции get. Подробнее о более лаконичных способах поиска без функции get рассказывается в разделе «Коллекции – это функции», ниже.

contains? Функция `contains?` – это предикат, возвращающий `true` для ассоциативной коллекции, только если в ней присутствует указанный ключ:

```
(contains? [1 2 3] 0)
;= true
(contains? {:a 5 :b 6} :b)
;= true
(contains? {:a 5 :b 6} 42)
;= false
(contains? #{1 2 3} 1)
;= true
```

> **Внимание.** Программисты на Clojure часто допускают ошибку, полагая, что `contains?` отыскивает в коллекции именно значение, то есть, считают, что эту функцию можно использовать для поиска определенного числового значения, в векторе, таком как `[0 1 2 3]`. Это заблуждение может приводить к весьма странным, на первый взгляд, результатам:
>
> ```
> (contains? [1 2 3] 3)
> ;= false
> (contains? [1 2 3] 2)
> ;= true
> (contains? [1 2 3] 0)
> ;= true
> ```
>
> Все результаты, полученные в этом примере, правильные, потому что `contains?` проверяет наличие ключа в указанной ассоциативной коллекции, то есть, в данном случае – наличие индексов `3`, `2` и `0`. Для проверки присутствия в коллекции определенного значения, обычно используется функция `some`, описываемая в разделе «Коллекции, ключи и функции высшего порядка», ниже.

Функции `get` и `contains?` наиболее универсальны: они эффективно работают с векторами, ассоциативными массивами, множествами, ассоциативными массивами Java, строками и массивами Java:

```
(get "Clojure" 3)
;= \j
(contains? (java.util.HashMap.) "not-there")
;= false
(get (into-array [1 2 3]) 0)
;= 1
```

Берегитесь значения nil

Когда в коллекции отсутствует искомый ключ и не указано значение по умолчанию, `get` возвращает `nil`. Однако `nil` является допустимым значением и может быть возвращено, если связано с указанным ключом:

```
(get {:ethel nil} :lucy)
;= nil
(get {:ethel nil} :ethel)
;= nil
```

Как же тогда отличить ключ, связанный со значением `nil`, от ситуации, когда ключ отсутствует в коллекции?

Можно было бы использовать `contains?`, но тогда каждый раз пришлось бы дважды осуществлять поиск в ассоциативном массиве: первый – чтобы с помощью `contains?` определить наличие искомого ключа, и второй – чтобы получить фактическое значение. Можно пойти на хитрость и указать в вызове `get` специальное значение по умолчанию, по которому можно было бы судить об отсутствии ключа. Однако этот прием имеет серьезные недостатки: во-первых, специальное значение легко может просочиться в ассоциативный массив, как нормальное значение ключа, а во-вторых, его неудобно использовать в условных выражениях, так как любое специальное значение по умолчанию в логическом контексте всегда будет интерпретироваться как истинное, даже когда искомый ключ отсутствует.

Данная проблема легко решается с помощью `find`. Функция `find` действует подобно функции `get`, за исключением того, что вместо ассоциированного значения она возвращает пару ключ/значение целиком или `nil`, если ключ не найден.

```
(find {:ethel nil} :lucy)
;= nil
(find {:ethel nil} :ethel)
;= [:ethel nil]
```

Кроме того, функция `find` отлично подходит для использования в условных формах и в формах деструктуризации, таких как `if-let` (или `when-let`):

```
(if-let [e (find {:a 5 :b 6} :a)]
  (format "found %s => %s" (key e) (val e))
  "not found")
;= "found :a => 5"
(if-let [[k v] (find {:a 5 :b 6} :a)]
  (format "found %s => %s" k v)
  "not found")
;= "found :a => 5"
```

Конечно, если достаточно просто проверить наличие некоторого ключа, можно использовать `contains?`.

Внимание. Берегитесь значения false. При использовании в условных выражениях, значения `false` из ассоциативных коллекций порождают аналогичную проблему, что и значения `nil`, и для ее решения необходимо предпринимать те же меры.

Индексирование

До сих пор, говоря о векторах, мы избегали обсуждения вопросов, касающихся извлечения произвольного *n*-го элемента или изменения его значения. Причина в том, что *индексы являются новыми указателями*[1].

Вообще говоря, индексы – в строках, массивах или в других последовательностях – редко действительно незаменимы в реализациях ваших алгоритмов[2]. Во многих случаях использование индексов лишь привносит дополнительные сложности: необходимость использования арифметики индексов, проверки границ и излишнюю косвенность. Исключая особые случаи, можно смело утверждать, что применение индексов для поиска или изменения значений является дурным тоном.

---

[1] А `IndexOutOfBoundsException` – новое исключение.

[2] Всякое обобщение имеет исключения. Если вы собираетесь написать библиотеку для численного анализа или реализовать сублинейные алгоритмы поиска, такие как алгоритм Бойера-Мура (Boyer-Moore) поиска подстроки в строке, индексы безусловно будут необходимы. Иногда, чтобы реализовать некоторый алгоритм на языке Clojure, имеет смысл отступить на шаг назад. Массивы и индексы используются во многих учебниках и руководствах просто потому, что они являются универсальным способом описания алгоритмов, своего рода общим знаменателем для различных языков программирования. Но это не означает, что в языке Clojure невозможны другие, более ясные и эффективные способы реализации.

Тем не менее, иногда индексы действительно бывают необходимы, и тогда нам на помощь приходит абстракция индексирования. Она представлена единственной функцией `nth`, являющейся специализированной версией `get`. Они отличаются реакцией на индексы, выходящие за границы коллекции: функция `nth` возбуждает исключение, а `get` возвращает `nil`:

Пример 3.3. Сравнение функций nth и get при работе с векторами

```
(nth [:a :b :c] 2)
;= :c
(get [:a :b :c] 2)
;= :c
(nth [:a :b :c] 3)
;= java.lang.IndexOutOfBoundsException
(get [:a :b :c] 3)
;= nil
(nth [:a :b :c] -1)
;= java.lang.IndexOutOfBoundsException
(get [:a :b :c] -1)
;= nil
```

Однако, при передаче значения по умолчания, семантика этих функций становится идентичной:

```
(nth [:a :b :c] -1 :not-found)
;= :not-found
(get [:a :b :c] -1 :not-found)
;= :not-found
```

Функции `nth` и `get` имеют разное назначение. Во-первых, `nth` может работать только с числовыми индексами и коллекциями, поддерживающими индексирование числами: векторами, списками, последовательностями, массивами Java, списками Java, строками и результатами сопоставления с регулярными выражениями. Функция `get`, напротив, является более универсальной: она может работать с любыми ассоциативными типами, как было показано выше, и интерпретирует числовые индексы как ключи в рассматриваемой коллекции или значении.

Другое важное отличие между `nth` и `get` заключается в том, что `get` является более устойчивой. Выше уже было показано, что в случае отсутствия индекса (интерпретируемого как ключ) `get` не возбуж-

дает исключение, а возвращает `nil`. Более того, `get` возвращает `nil`, даже когда целевое значение не поддерживает семантику поиска, а функция `nth` возбуждает исключение.

```
(get 42 0)
;= nil
(nth 42 0)
;= java.lang.UnsupportedOperationException: nth not supported on this type:
;= Long
```

> **Примечание.** Векторы в Clojure поддерживают семантику `nth` непосредственно и могут использоваться без функции `nth`. Подробнее о более лаконичных способах поиска без функции `nth` рассказывается в разделе «Коллекции – это функции», ниже.

Стек

Стек – это коллекция, поддерживающая семантику «последним пришел, первым ушел (Last-In First-Out, LIFO), то есть, первым из стека будет извлечен элемент, помещенный в него последним. В языке Clojure отсутствует отдельная структура данных, реализующая стек, но он обладает поддержкой абстракции стека в виде трех операций:

- `conj` – вталкивает значение в стек (отличный пример повторного использования обобщенной операции над коллекциями);
- `pop` – выталкивает значение с вершины стека;
- `peek` – позволяет получить значение с вершины стека, не выталкивая его.

В роли стеков могут использоваться списки и векторы (примеры 3.4 и 3.5), в которых роль вершины стека играет тот конец структуры данных, с которым наиболее эффективно работает функция `conj`.

Пример 3.4. Использование списка в качестве стека

```
(conj '() 1)
;= (1)
(conj '(2 1) 3)
;= (3 2 1)
(peek '(3 2 1))
;= 3
(pop '(3 2 1))
```

```
;= (2 1)
(pop '(1))
;= ()
```

Пример 3.5. Использование вектора в качестве стека

```
(conj [] 1)
;= [1]
(conj [1 2] 3)
;= [1 2 3]
(peek [1 2 3])
;= 3
(pop [1 2 3])
;= [1 2]
(pop [1])
;= []
```

Применение `pop` к пустому стеку вызывает ошибку.

Множество

Выше уже демонстрировалось, что множества поддерживают абстракцию ассоциативных коллекций и могут интерпретироваться как разновидность вырожденных ассоциативных массивов, в которых ключи ассоциированы сами с собой:

```
(get #{1 2 3} 2)
;= 2
(get #{1 2 3} 4)
;= nil
(get #{1 2 3} 4 "not-found")
;= "not-found"
```

Однако абстракция *множества* дополнительно поддерживает операцию `disj` – удаления значения (или нескольких значений) из указанного множества:

```
(disj #{1 2 3} 3 1)
;= #{2}
```

Абстракция множества сама по себе не слишком велика и большинство основных операций над множествами поддерживают общую семантику коллекций и ассоциативных коллекций, тем не

менее, мы рекомендуем ознакомиться с описанием `clojure.set`. Пространство имен `clojure.set` в стандартной библиотеке Clojure содержит функции, реализующие некоторые высокоуровневые операции с множествами и предикаты, включая `subset?`, `superset?`, `union`, `intersection`, `project` и другие.

Сортированные коллекции

Коллекции, поддерживающие абстракцию *сортированных коллекций*, гарантируют расположение значений в них в некотором порядке, который может определяться предикатом или реализацией специального интерфейса сравнения. Это позволяет получать последовательности значений, упорядоченных в прямом или обратном порядке, по всему диапазону или по его части. Данная абстракция поддерживается следующими операциями:

- `rseq` – возвращает последовательность значений коллекции, расположенных в обратном порядке, при этом гарантируется, что время выполнения операции не будет зависеть от величины коллекции;
- `subseq` – возвращает последовательность значений коллекции из указанного диапазона ключей;
- `rsubseq` – то же, что и `subseq`, но значения в последовательности расположены в обратном порядке.

Роль сортированных коллекций могут играть только ассоциативные массивы и множества. Они не имеют литерального представления; могут создаваться с помощью `sorted-map` и `sorted-set`, или `sorted-map-by` и `sorted-set-by`, если необходимо определить свой предикат или реализацию сравнения для сортировки.

К сортированной коллекции могут применяться любые функции поддержки абстракции:

```
(def sm (sorted-map :z 5 :x 9 :y 0 :b 2 :a 3 :c 4))
;= #'user/sm
sm
;= {:a 3, :b 2, :c 4, :x 9, :y 0, :z 5}
(rseq sm)                                          ❶
;= ([:z 5] [:y 0] [:x 9] [:c 4] [:b 2] [:a 3])
(subseq sm <= :c)                                  ❷
;= ([:a 3] [:b 2] [:c 4])
(subseq sm > :b <= :y)                             ❸
;= ([:c 4] [:x 9] [:y 0])
```

```
(rsubseq sm > :b <= :y)                                    ❹
;= ([:y 0] [:x 9] [:c 4])
```

❶ `rseq` возвращает последовательность с содержимым `sm`, расположенным в обратном порядке, и гарантирует, что время переупорядочения не будет зависеть от длины исходной коллекции.
❷ Здесь запрашивается часть элементов коллекции `sm`, ключи в которых меньше или равны `:c`.
❸ Этот запрос отыскивает все элементы с ключами больше `:b` и меньше или равными `:y`.
❹ `rsubseq` действует так же, как и `subseq`, но элементы в возвращаемой последовательности расположены в обратном порядке.

Так как `sm` является сортированной коллекцией, каждая из этих операций обладает гораздо более высокой производительностью, чем аналогичные им обобщенные операции над обычными последовательностями (такие как `filter`, `take-while`), производительность которых имеет линейную зависимость от длины последовательности. В частности, `rseq` гарантирует постоянное время выполнения, тогда как время выполнения `reverse`, которая может использоваться для получения последовательности элементов из любой коллекции, расположенных в обратном порядке, линейно зависит от длины коллекции[1].

Порядок сортировки по умолчанию – по возрастанию – определяется функцией `compare`, которая поддерживает все скаляры языка Clojure и упорядоченные коллекции, и обеспечивает лексикографическую сортировку на каждом уровне[2]:

```
(compare 2 2)
;= 0
(compare "ab" "abc")
;= -1
(compare ["a" "b" "c"] ["a" "b"])
;= 1
(compare ["a" 2] ["a" 2 0])
;= -1
```

[1] Здесь мы немного грешим против истины: формально `rseq` является частью другой небольшой абстракции – *обратимых коллекций* (reversible) – дающей такую гарантию. Мы умолчали об этом потому, что помимо сортированных коллекций только векторы являются обратимыми (и поэтому `rseq` может также применяться к векторам).

[2] Из этого следует, например, что с ее помощью можно отсортировать векторы векторов из векторов.

В действительности функция compare поддерживает не только строки, числа и упорядоченные коллекции: она поддерживает все, что реализует интерфейс java.lang.Comparable, включая логические значения, ключевые слова, символы и все Java-классы, реализующие этот интерфейс. compare является довольно мощной функцией, но она – всего лишь компаратор (comparator), используемый по умолчанию.

Определение порядка с помощью компараторов и предикатов

Компаратор (comparator) – это функция с двумя аргументами, возвращающая положительное целое, если первый аргумент больше второго, отрицательное – если первый аргумент меньше второго, и ноль – если аргументы равны.

Все функции в языке Clojure реализуют интерфейс java.util.Comparator и потому могут использоваться в роли компараторов, но не все они предназначены для применения в этом качестве. Чтобы превратить функцию в реализацию интерфейса Comparator, не требуется добавлять какие-то специальные определения – любой предикат с двумя аргументами автоматически реализует его.

Помимо отсутствия необходимости реализовать специальный интерфейс, чтобы создать компаратор, этот факт означает также, что определение довольно сложных правил упорядочения легко можно выразить, используя прием композиции функций. Функции сравнения можно передавать непосредственно функциям создания сортированных коллекций, а также функциям sort и sort-by[1]:

```
(sort < (repeatedly 10 #(rand-int 100)))
;= (12 16 22 23 41 42 61 63 83 87)
(sort-by first > (map-indexed vector "Clojure"))
;= ([6 \e] [5 \r] [4 \u] [3 \j] [2 \o] [1 \l] [0 \C])
```

Таким образом, функции sorted-map и sorted-set создают ассоциативные массивы и множества, ключи в которых сортируются с применением функции compare, а sorted-map-by и sorted-set-by позволяют указать свой компаратор (то есть, любую функцию-предикат с двумя аргументами), определяющий порядок сортировки. Простейшим компаратором, который можно передать функции создания сортированной коллекции (кроме самой функции compare), является, по-

[1] Или любому Java API, принимающему реализацию java.util.Comparator.

Как в Clojure предикат превращается в компаратор?

Кажется необычным, что предикаты, такие как <, возвращающие логическое значение, могут использоваться в роли компараторов, которые должны возвращать положительное или отрицательное целое, или ноль, если аргументы равны.

Алгоритм превращения предиката в компаратор прост: если при первом вызове предиката с двумя аргументами в указанном порядке возвращается значение `true`, оно преобразуется в `-1`. Иначе предикат вызывается еще раз, но с обратным порядком следования аргументов. Если на этот раз будет возвращено значение `true`, оно преобразуется в `1`. В противном случае считается, что аргументы равны и в качестве окончательного результата возвращается `0`.

Функция `comparator` явно преобразует предикат с двумя аргументами в функцию-компаратор, реализующую следующую логику:

```
((comparator <) 1 4)
;= -1
((comparator <) 4 1)
;= 1
((comparator <) 4 4)
;= 0
```

Она редко используется на практике, потому что функции неявно обеспечивают необходимые преобразования при использовании их с различными функциями, принимающими компараторы, и функции с двумя аргументами уже реализуют интерфейс `java.util.Comparator`.

жалуй, выражение `(comp - compare)`, инвертирующее результат `compare` и, соответственно, порядок сортировки:

```
(sorted-map-by compare :z 5 :x 9 :y 0 :b 2 :a 3 :c 4)
;= {:a 3, :b 2, :c 4, :x 9, :y 0, :z 5}
(sorted-map-by (comp - compare) :z 5 :x 9 :y 0 :b 2 :a 3 :c 4)
;= {:z 5, :y 0, :x 9, :c 4, :b 2, :a 3}
```

Следует отметить, что *порядок сортировки определяет понятие равенства внутри сортированного ассоциативного массива или множества*; иногда это может приводить к неожиданным, но вполне объяснимым результатам. Например, допустим, что имеется функция, возвращающая порядок величины числа:

```
(defn magnitude
  [x]
  (-> x Math/log10 Math/floor))
;= #'user/magnitude
(magnitude 100)
;= 2.0
(magnitude 100000)
;= 5.0
```

На ее основе достаточно просто можно создать предикат сравнения, возвращающий разность порядков первого и второго аргументов:

```
(defn compare-magnitude
  [a b]
  (- (magnitude a) (magnitude b)))
((comparator compare-magnitude) 10 10000)
;= -1
((comparator compare-magnitude) 100 10)
;= 1
((comparator compare-magnitude) 10 75)
;= 0
```

Если теперь использовать эту функцию как компаратор для сортированной коллекции, возникает любопытная проблема:

```
(sorted-set-by compare-magnitude 10 1000 500)   ❶
;= #{10 500 1000}
(conj *1 600)                                   ❷
;= #{10 500 1000}
(disj *1 750)                                   ❸
;= #{10 1000}
(contains? *1 1239)                             ❹
;= true
```

❶ Каждому значению, 10, 1000 и 500 соответствует свой порядок величины, поэтому они сохраняются в множестве, как отличные друг от друга элементы, согласно результату, возвращаемому компаратором.

❷ Попытка добавить число 600 игнорируется, потому что число 600 имеет тот же порядок величины, что и число 500, и потому компаратором эти значения считаются одинаковыми.

❸ Поскольку с точки зрения компаратора число 750 также считается равным числу 500, последнее удаляется из множества, даже при том,

что число 750 было передано как аргумент функции `disj`, не имеющей отношения к сортировке.

❹ Аналогично, число 1239 имеет тот же порядок величины, что и число 1000, `contains?` возвращает для него `true`, обнаружив в данном множестве ключ, равный указанному значению.

Иногда такое поведение бывает желательным, иногда – нет. Не забывайте, что реализация компараторов полностью находится в вашей власти; несмотря на удобство использования (или повторного использования) предикатов, в любой момент можно организовать возврат отрицательных и положительных целых чисел или нуля, чтобы обеспечить желаемую семантику равенства. Функцию `compare-magnitude` можно переписать так, чтобы равными считались только действительно равные значения:

```
(defn compare-magnitude
  [a b]
  (let [diff (- (magnitude a) (magnitude b))]
    (if (zero? diff)
      (compare a b)
      diff)))

(sorted-set-by compare-magnitude 10 1000 500)
;= #{10 500 1000}
(conj *1 600)
;= #{10 500 600 1000}
(disj *1 750)
;= #{10 500 600 1000}
```

Теперь значения в множестве будут отсортированы по в соответствии с порядком их величин, но операции, опирающиеся на понятие равенства (такие как `conj` и `disj`), будут действовать более предсказуемо.

Функции `subseq` и `rsubseq` будут действовать с сортированными коллекциями ожидаемым образом, извлекая интервалы (в прямом и обратном порядке сортировки) в полном соответствии с указанным компаратором:

```
(sorted-set-by compare-magnitude 10 1000 500 670 1239)
;= #{10 500 670 1000 1239}
(def ss *1)
;= #'user/ss
```

```
(subseq ss > 500)
;= (670 1000 1239)
(subseq ss > 500 <= 1000)
;= (670 1000)
(rsubseq ss > 500 <= 1000)
;= (1000 670)
```

Примечание. Операторы <, <=, > и >=, определяющие интервалы в этих функциях, используется исключительно как подсказки, фактическое же сравнение производится с помощью указанного компаратора – предикаты, соответствующие этим операторам, не используются.

Одним из интересных применений этих функций является реализация линейной интерполяции:

```
(defn interpolate
 "Принимает коллекцию координат точек (в виде кортежей [x y]) и возвращает
  Функцию, обеспечивающую линейную интерполяцию между этими точками."
  [points]
  (let [results (into (sorted-map) (map vec points))]   ❶
    (fn [x]
      (let [[xa ya] (first (rsubseq results <= x))      ❷
            [xb yb] (first (subseq results > x))]
        (if (and xa xb)                                 ❸
          (/ (+ (* ya (- xb x)) (* yb (- x xa)))        ❹
             (- xb xa))
          (or ya yb))))))
```

❶ (map vec points) проверяет, является ли каждая точка вектором и добавляет их в ассоциативный массив.

❷ Здесь и в следующей строке среди имеющихся точек отыскиваются две, ближайшие к указанной координате x.

❸ При выходе за границы области определений, когда xa или xb получает значение nil, возвращается (or ya yb), являющееся единственным известным значением.

❹ Формула линейной интерполяции для обычного случая.

Проверим эту функцию на примере трех известных точек, [0 0], [10 10] и [15 5] (рис. 3.3).

По известно координате x можно определить значение y, соответствующее имеющимся данным:

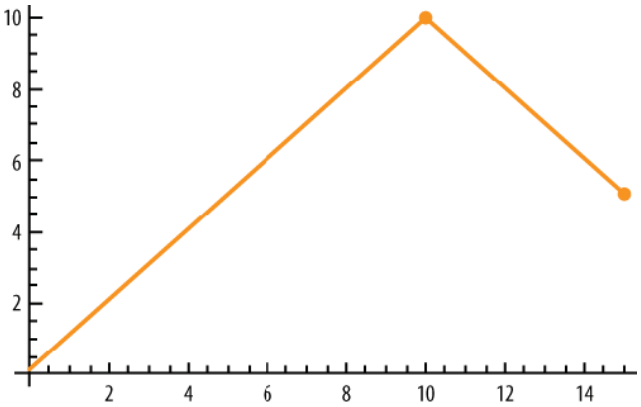

Рис. 3.3. Три известные точки

```
(def f (interpolate [[0 0] [10 10] [15 5]]))
;= #'user/f
(map f [2 10 12])
;= (2 10 8)
```

Отлично!

Упрощенный доступ к коллекциям

Доступ к значениям вполне можно назвать самой распространенной операцией над коллекциями. Это особенно справедливо для ассоциативных коллекций. В таких ситуациях необходимость постоянно вводить имена функций `get` и `nth` может оказаться весьма утомительной. К счастью коллекции в Clojure и наиболее часто используемые типы ключей, допустимые в ассоциативных коллекциях, *являются одновременно функциями*, реализующими семантику функции `get` или `nth` (в зависимости от конкретного типа коллекции).

Коллекции – это функции. Все просто – коллекции в Clojure являются функциями, отыскивающими значение, связанное с указанным ключом или индексом. То есть, следующие строки:

```
(get [:a :b :c] 2)
;= :c
(get {:a 5 :b 6} :b)
;= 6
```

```
(get {:a 5 :b 6} :c 7)
;= 7
(get #{1 2 3} 3)
;= 3
```

в точности эквивалентны следующим, более кратким выражениям:

```
([:a :b :c] 2)
;= :c
({:a 5 :b 6} :b)
;= 6
({:a 5 :b 6} :c 7)
;= 7
(#{1 2 3} 3)
;= 3
```

В каждом из этих выражений коллекция располагается в позиции функции, и *она* вызывается с ключом или индексом, который требуется отыскать. Ассоциативные массивы могут принимать второй необязательный аргумент, подобно функции `get`, – значение по умолчанию, возвращаемое в случае неудачного поиска. Коллекции двух других типов, векторы и множества, принимают только одно значение/индекс – они не поддерживают значения по умолчанию. Кроме того, индексы, передаваемые вектору для поиска, также должны входить в число имеющихся индексов, как и в функции `nth`:

```
([:a :b :c] -1)
;= #<IndexOutOfBoundsException java.lang.IndexOutOfBoundsException>
```

Ключи коллекций являются функциями (часто). Аналогично, наиболее часто используемые типы ключей – ключевые слова и символы – также являются функциями, осуществляющими поиск в указанных коллекциях. То есть, следующие строки:

```
(get {:a 5 :b 6} :b)
;= 6
(get {:a 5 :b 6} :c 7)
;= 7
(get #{:a :b :c} :d)
;= nil
```

в точности эквивалентны более кратким выражениям:

```
(:b {:a 5 :b 6})
;= 6
(:c {:a 5 :b 6} 7)
;= 7
(:d #{:a :b :c})
;= nil
```

Так как значение в позиции функции должно быть функцией, этот прием нельзя применять к числовым индексам, то есть, поиск в векторе не может быть выполнен таким способом.

Идиоматические приемы использования

Итак, в нашем распоряжении имеются более лаконичные способы доступа к значениям в коллекциях. Это хорошо, но необходимо дополнительно прояснить, когда и как использовать разные вариации: когда в роли функции следует использовать коллекцию, а когда ключевое слово или символ.

Вступая сюда, мы попадаем в опасную область личных предпочтений, но *в общем случае* можно порекомендовать использовать в качестве функций ключевые слова или символы. Наиболее очевидным преимуществом этой идиомы является возможность избежать исключений, связанных с использованием пустого указателя, потому что ключевые слова и символы чаще всего представлены в виде литералов, при использовании их в роли функций. Сравните:

```
(defn get-foo
  [map]
  (:foo map))
;= #'user/get-foo
(get-foo nil)
;= nil

(defn get-bar
  [map]
  (map :bar))
;= #'user/get-bar
(get-bar nil)
;= #<NullPointerException java.lang.NullPointerException>
```

Кроме того, форма `(coll :foo)` предполагает, что `coll`, коллекция, одновременно является функцией. Это верно для большинства

структур данных в языке Clojure, но не верно (например) для списков, и не обязательно верно для других типов, поддерживающих абстракцию коллекций, которые могут и не быть функциями. Все это делает форму `(:foo coll)` более предпочтительной, поскольку она гарантирует, что ключ `:foo` всегда является функцией и никогда не будет иметь значение `nil`, тогда как значение, на которое ссылается `coll`, может не удовлетворять этим условиям.

Конечно, при использовании ключей других типов, не являющихся ключевыми словами или символами, в роли функции придется использовать сами коллекции или задействовать функцию `get` или `nth`.

Коллекции, ключи и функции высшего порядка

Так как ключевые слова, символы и многие коллекции являются функциями, использование их в качестве аргументов функций высшего порядка не только вполне обычно, но и невероятно удобно. Допустим, нам потребовалось извлечь имена всех клиентов. Для этого не нужно определять новую функцию или явно использовать `get`:

```
(map :name [{:age 21 :name "David"}
            {:gender :f :name "Suzanne"}
            {:name "Sara" :location "NYC"}])
;= ("David" "Suzanne" "Sara")
```

Функция `some` отыскивает первое значение в последовательности, для которого указанный предикат вернет логически истинное значение. Она часто используется для поиска в множествах:

```
(some #{1 3 7} [0 2 4 5 6])
;= nil
(some #{1 3 7} [0 2 3 4 5 6])
;= 3
```

Это делает функцию `some` очень удобной для проверки результатов поиска в условных выражениях. Еще более обобщенной является функция `filter`, которая возвращает «ленивую» последовательность значений, соответствующих указанному предикату. Ей так же можно передать коллекцию, ключевое слово или символ, возможно в комбинации с другими функциями:

```
(filter :age [{:age 21 :name "David"}
              {:gender :f :name "Suzanne"}
              {:name "Sara" :location "NYC"}])
;= ({:age 21, :name "David"})

(filter (comp (partial <= 25) :age) [{:age 21 :name "David"}
                                     {:gender :f :name "Suzanne" :age 20}
                                     {:name "Sara" :location "NYC" :age 34}])
;= ({:age 34, :name "Sara", :location "NYC"})
```

Функция `remove` является дополнением к функции `filter` в буквальном смысле слова: она реализует фильтрацию коллекции, применяя функцию `complement` к указанной функции, `(filter (complement f) collection)`.

> **Внимание. Берегитесь значения nil (еще раз).** Множества настолько просто использовать для проверки присутствия в коллекции некоторого значения, что легко забыть об особенностях значений `nil` и `false`, которые в логическом контексте интерпретируются как ложные, в противоположность нашим ожиданиям:
>
> ```
> (remove #{5 7} (cons false (range 10)))
> ;= (false 0 1 2 3 4 6 8 9)
> (remove #{5 7 false} (cons false (range 10)))
> ;= (false 0 1 2 3 4 6 8 9)
> ```
>
> Поэтому, если заранее неизвестно, могут ли присутствовать значения `nil` или `false` в множестве, вместо `get` или самого множества в роли предиката лучше использовать `contains?`:
>
> ```
> (remove (partial contains? #{5 7 false}) (cons false (range 10)))
> ;= (0 1 2 3 4 6 8 9)
> ```

Типы структур данных

В языке Clojure имеется множество конкретных структур данных, удовлетворяющих различным абстракциям. Мы уже неоднократно сталкивались с этими конкретными реализациями – их поведение и семантика в значительной степени определяется поддерживаемыми ими абстракциями.

В этом разделе мы коротко пройдемся по некоторым отличительным особенностям реализаций конкретных типов структур данных, большинство из которых связаны с конструктивными особенностями.

Списки

Списки – простейший тип коллекций в Clojure. Основное их назначение – служить представлением вызовов функций в программном коде, как описывалось в разделе «Выражения, операторы, синтаксис и очередность», в главе 1. Списки намного чаще используются в виде литералов в исходном коде, чем во время выполнения[1].

Списки в языке Clojure являются односвязными, но высокая эффективность доступа или «изменения» обеспечивается только для головы списка, с применением функции `conj`, добавляющей новое значение в начало списка, а также `pop` и оператора `rest`, возвращающего ссылку на подсписок, следующий за первым значением в исходном списке. Связанные списки не поддерживают произвольный доступ, поэтому время выполнения функции `nth` со списком прямо пропорционально положению элемента в списке (в противоположность постоянному времени доступа при работе с векторами, массивами и другими коллекциями). А функция `get` вообще не поддерживает списки, потому что в противном случае она не обеспечивала бы гарантированную эффективность.

Стоит также отметить, что списки сами являются последовательностями, поэтому если передать функции `seq` список, она вернет сам этот список, а не отдельное последовательное представление списка.

Выше мы уже видели литералы списков:

```
'(1 2 3)
;= (1 2 3)
```

Если опустить апостроф перед списком, он будет интерпретироваться как вызов значения `1`, что вызовет ошибку[2]. Как побочный эффект такой формы записи, выражения внутри литерала списка также не вычисляются:

---

[1] Это является отступлением от традиций, свойственных прежним диалектам Lisp, где списки и cons-ячейки играют главную роль. В Clojure используются более богатые возможностями структуры данных, такие как ассоциативные массивы, множества, векторы и абстрактные родственники списков – последовательности, что существенно снижает потребность в применении списков как таковых.

[2] Обратите внимание, что апостроф перед пустым списком можно опустить, потому что в нем отсутствует первый элемент, который мог бы рассматриваться как вызываемое значение. То есть, `()` – это допустимый литерал пустого списка.

```
'(1 2 (+ 1 2))
;= (1 2 (+ 1 2))
```

В таких случаях большинство используют литерал вектора, внутри которого члены-выражения всегда вычисляются. Однако в некоторых ситуациях необходимы именно списки, а не какие-то другие структуры данных[1]. И тогда можно воспользоваться функцией `list`:

```
(list 1 2 (+ 1 2))
;= (1 2 3)
```

Функция `list` принимает произвольное количество значений, из которых каждое становится элементом возвращаемого списка.

Наконец, для проверки присутствия значения в списке можно использовать предикат `list?`.

Векторы

Векторы являются последовательными структурами данных, поддерживающими эффективные операции произвольного доступа и изменения элементов, соответствующие ожиданиям программистов, использующих `java.util.ArrayList`, списки в Python и массивы в Ruby. Кроме того векторы отличаются особой гибкостью, особенно в отношении абстракций ассоциативных коллекций, индексирования и стеков, как уже было показано выше.

Помимо использования хорошо знакомых литералов, векторы можно также создавать с помощью `vector` и `vec`:

```
(vector 1 2 3)
;= [1 2 3]
(vec (range 5))
;= [0 1 2 3 4]
```

Функция `vector` является аналогом функции `list`, тогда как `vec` принимает единственный аргумент-последовательность, содержимое которого целиком будет использовано для создания вектора. Это удобно, когда имеются какие-то данные в виде массива, списка, последовательности или другого значения, которое может быть пре-

---

[1] Наиболее типичными такими ситуациями является создание макросов, о которых рассказывается в главе 5.

образовано в последовательность, но для дальнейших манипуляций с данными необходимы некоторые особенности, присущие векторам.

Функция `vector?` – это предикат, аналог `list?`, используемый для проверки наличия некоторого значения в векторе.

Векторы как кортежи

Кортежи (tuples) – это одна из наиболее часто используемых разновидностей векторов. Всякий раз, когда требуется объединить несколько значений с минимумом церемоний – например, чтобы вернуть несколько значений из функции – их можно поместить в вектор:

```
(defn euclidian-division    ❶
  [x y]
  [(quot x y) (rem x y)])
(
euclidian-division 42 8)
;= [5 2]
```

❶ Просто для тех, кому любопытно, выражение `(juxt quot rem)` вернет функцию, эквивалентную данной.

Кортежи отлично сочетаются с вездесущим механизмом деструктуризации (см. «Деструктуризация (let, часть 2)» в главе 1), позволяющим легко распаковывать такие возвращаемые значения на составляющие:

```
(let [[q r] (euclidian-division 53 7)]
  (str ¨53/7 = ¨ q ¨ * 7 + ¨ r))
;= ¨53/7 = 7 * 7 + 4¨
```

Несмотря на заманчивость и простоту кортежей, не следует забывать, что они не самый лучший механизм передачи данных: их лучше хранить скрытыми в недрах библиотек и модулей и не выставлять как часть общедоступного API. На то есть две причины:

- кортежи не являются самодокументируемыми – вам придется постоянно вспоминать назначение каждого индекса;
- кортежи не отличаются гибкостью – вы должны будете указывать значения для элементов в середине, даже если они не имеют смысла для данного возвращаемого значения, и вы не сможете расширить кортеж иначе, чем добавлять новые элементы в конец.

Ассоциативные массивы не страдают этими недостатками, поэтому они лучше подходят для функций, являющихся частью общедоступного API, и непростых возвращаемых значений.

Однако всякое правило имеет исключения. В предметных областях, где назначение кортежей очевидно, – для представления координат, ориентированных ребер графов и так далее – применение векторов в виде кортежей вполне оправданно:

```
(def point-3d [42 26 -7])

(def travel-legs [["LYS" "FRA"] ["FRA" "PHL"] ["PHL" "RDU"]])
```

Обратите внимание, что в последнем примере выше векторы играют две разные роли: внешний вектор используется в качестве списка, а внутренние действуют как кортежи с трехбуквенными обозначениями аэропортов, `[from to]`.

Множества

О множествах, как о конкретной реализации структур данных, мало что можно сказать в дополнение к тому, что уже говорилось при обсуждении абстракций, таких как ассоциативные коллекции и множества. Как и другие типы структур данных в Clojure, множества имеют литеральное представление, которое мы уже видели:

```
#{1 2 3}
;= #{1 2 3}
#{1 2 3 3}      ❶
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   Duplicate key: 3>
```

❶ Множества не могут содержать повторяющиеся значения по определению, поэтому литералы с повторяющимися значениями вызывают ошибку.

Создать несортированное множество можно с помощью функции `hash-set`, принимающей произвольное количество аргументов:

```
(hash-set :a :b :c :d)
;= #{:a :c :b :d}
```

Наконец, множество можно создать из любой коллекции с помощью функции `set`:

```
(set [1 6 1 8 3 7 7])
;= #{1 3 6 7 8}
```

Этот прием можно применить к любому значению, пригодному для преобразования в последовательность, и позволяет использовать весьма выразительные идиомы, основанные на том, что множества сами являются функциями:

```
(apply str (remove (set "aeiouy") "vowels are useless"))
;= "vwls r slss"

(defn numeric? [s] (every? (set "0123456789") s))
;= #'user/numeric?
(numeric? "123")
;= true
(numeric? "42b")
;= false
```

Доступны также сортированные разновидности множеств, которые демонстрировались в разделе «Сортированные коллекции», выше.

Ассоциативные массивы

Помимо знакомых уже литералов ассоциативных массивов:

```
{:a 5 :b 6}
;= {:a 5, :b 6}
{:a 5 :a 5}                ❶
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   Duplicate key: :a>
```

❶ Как и значения в множествах, ключи в ассоциативных массивах должны быть уникальными. Литералы, нарушающие это требование, вызывают ошибку.

Несортированные ассоциативные массивы можно создавать с помощью функции `hash-map`, принимающей произвольное количество пар ключ/значение. Чаще всего она используется в комбинации с функцией `apply`, когда имеется коллекция пар ключ/значение, не сгруппированных в вектор кортежей:

```
(hash-map :a 5 :b 6)
;= {:a 5, :b 6}
(apply hash-map [:a 5 :b 6])
;= {:a 5, :b 6}
```

Имеется также возможность создавать сортированные ассоциативные массивы, как описывалось в разделе «Сортированные коллекции», выше.

Функции keys и vals. Эти функции работают только с ассоциативными массивами, тем не менее, их удобно использовать для получения последовательностей ключей или значений из исходного ассоциативного массива:

```
(keys m)
;= (:a :b :c)
(vals m)
;= (1 2 3)
```

По сути они являются более краткими вариантами получения из ассоциативного массива последовательности записей с помощью функции `seq` с последующим извлечением ключей или значений:

```
(map key m)
;= (:a :c :b)
(map val m)
;= (1 3 2)
```

Ассоциативные массивы как специализированные структуры

Ассоциативные массивы способны хранить значения любых типов, поэтому их часто используют как простые, гибкие структуры, где в качестве ключей используются ключевые слова, отождествляющие поля (также называются *слотами* (slots)).

```
(def playlist
  [{:title "Elephant", :artist "The White Stripes", :year 2003}
   {:title "Helioself", :artist "Papas Fritas", :year 1997}
   {:title "Stories from the City, Stories from the Sea",
    :artist "PJ Harvey", :year 2000}
   {:title "Buildings and Grounds", :artist "Papas Fritas", :year 2000}
   {:title "Zen Rodeo", :artist "Mardi Gras BB", :year 2002}])
```

Очень часто моделирование данных на языке Clojure начинается с создания простых ассоциативных массивов. Ассоциативные массивы позволяют приступить к работе немедленно, не вынуждая определять фиксированные модели, что особенно ценно, когда перечень слотов заранее неизвестен.

При использовании ассоциативных массивов (и других структур данных, поддерживающих абстракции языка Clojure) для нужд моделирования, вы получаете в свое распоряжение все возможности, которыми они обладают. Например, допустим, что в качестве ключей используются ключевые слова, тогда «запрос» данных из их совокупности превращается в тривиальную операцию:

```
(map :title playlist)
;= ("Elephant" "Helioself" "Stories from the City, Stories from the Sea"
;= "Buildings and Grounds" "Zen Rodeo")
```

Аналогично можно использовать такие возможности, как деструктуризация ассоциативных массивов, представленная в разделе «Деструктуризация ассоциативных массивов», в главе 1, упрощающие операции с отдельными «элементами» и устраняющие необходимость использовать более многословные способы доступа вида `(:slot data)`:

```
(defn summarize [{:keys [title artist year]}]
  (str title " / " artist " / " year))
```

Clojure старается уберечь вас от необходимости заранее создавать фиксированные модели данных, предлагая простой и понятный путь эволюционного развития – от простого прототипа на основе ассоциативного массива до более зрелой модели. То есть, модель, основанная на ассоциативном массиве, не является фиксированной. Напротив, пока вместо конкретных реализаций в программе используются абстракции коллекций, вы сохраняете возможность в любой момент заменить модель на основе ассоциативного массива более специализированной версией, например, объявленной с помощью формы `defrecord`, которая всегда производит ассоциативные типы.

Подробнее форма `defrecord` будет обсуждаться в разделе «Определение собственных типов», в главе 6, а создаваемые с ее помощью записи будут сравниваться с ассоциативными массивами в разделе «Когда использовать ассоциативные массивы, а когда записи», в главе 6.

Другие применения ассоциативных массивов

Ассоциативные массивы также часто используются для хранения сводной информации, индексов или таблиц преобразования (здесь подразумеваются аналоги индексов и представлений в базах данных).

Например, функцию `group-by` очень удобно использовать для разделения коллекций по ключевой функции:

```
(group-by #(rem % 3) (range 10))
;= {0 [0 3 6 9], 1 [1 4 7], 2 [2 5 8]}
```

Здесь числа сгруппированы по ключам, определяемым указанной функцией. Возвращаясь к примеру со списком произведений (`playlist`), с помощью этой функции легко можно создать оглавление альбома по имени исполнителя:

```
(group-by :artist playlist)
;= {"Papas Fritas" [{:title "Helioself", :artist "Papas Fritas", :year 1997}
;=                  {:title "Buildings and Grounds", :artist "Papas Fritas"}]
;= ...}
```

Индексирование по двум «столбцам» выполняется ничуть не сложнее: `(group-by (juxt :col1 :col2) data)`.

Иногда требуется получить суммарные значения по указанному ключу. Для этого можно было бы воспользоваться функцией `group-by` и затем просуммировать каждое значение:

```
(into {} (for [[k v] (group-by key-fn coll)]
             [k (summarize v)]))
```

...где на место `key-fn` и `summarize` необходимо подставить фактические функции. Однако с ростом размеров коллекций обрабатывать их становится все труднее. В этом случае можно создать собственную комбинацию из функций `group-by` и `reduce`. Такая функция `reduce-by` может использоваться для вычисления самых разнообразных суммарных значений данных, мало чем отличаясь от запроса на языке SQL: `SELECT ... GROUP BY ...`:

```
(defn reduce-by
  [key-fn f init coll]
  (reduce (fn [summaries x]
```

```
            (let [k (key-fn x)]
              (assoc summaries k (f (summaries k init) x))))
      {} coll))
```

> **Примечание. x, xs и другие незамысловатые имена.** Начинающие программировать на Clojure часто находят такие короткие имена, как `x` и `xs` слишком запутывающими. Библиотека стандартов оформления кода (http://dev.clojure.org/display/design/Library+Coding+Standards) определяет особые значения для некоторых коротких имен. По сути, используя короткое имя, такое как `x`, вы сообщаете об обобщенности своего кода и отсутствии необходимости учитывать его тип. Аналогично коллекциям или последовательностям дается имя `xs`, и так далее. Чем более универсальным является программный код, тем менее специфические имена в нем используются.

Представьте, что у нас имеется список заказов на поставку продуктов компании ACME Corp, представленный в виде коллекции простых ассоциативных массивов:

```
(def orders
  [{:product "Clock", :customer "Wile Coyote", :qty 6, :total 300}
   {:product "Dynamite", :customer "Wile Coyote", :qty 20, :total 5000}
   {:product "Shotgun", :customer "Elmer Fudd", :qty 2, :total 800}
   {:product "Shells", :customer "Elmer Fudd", :qty 4, :total 100}
   {:product "Hole", :customer "Wile Coyote", :qty 1, :total 1000}
   {:product "Anvil", :customer "Elmer Fudd", :qty 2, :total 300}
   {:product "Anvil", :customer "Wile Coyote", :qty 6, :total 900}])
```

С помощью `reduce-by` можно легко вычислить общую стоимость всех заказов для каждого заказчика:

```
(reduce-by :customer #(+ %1 (:total %2)) 0 orders)
;= {"Elmer Fudd" 1200, "Wile Coyote" 7200}
```

Аналогично можно получить список заказчиков для каждого продукта:

```
(reduce-by :product #(conj %1 (:customer %2)) #{} orders)
;= {"Anvil" #{"Wile Coyote" "Elmer Fudd"},
;=  "Hole" #{"Wile Coyote"},
;=  "Shells" #{"Elmer Fudd"},
;=  "Shotgun" #{"Elmer Fudd"},
;=  "Dynamite" #{"Wile Coyote"},
;=  "Clock" #{"Wile Coyote"}}
```

А что если потребуется получить двухуровневую сводную информацию, например, отобрать все заказы по заказчикам, а затем сгруппировать их по продукту? Для этого достаточно в качестве ключа вернуть вектор с двумя значениями. Написать такую функцию можно несколькими разными способами:

```
(fn [order]
  [(:customer order) (:product order)])

#(vector (:customer %) (:product %))

(fn [{:keys [customer product]}]
  [customer product])

(juxt :customer :product)
```

Выберем самую простую и короткую из них:

```
(reduce-by (juxt :customer :product)
  #(+ %1 (:total %2)) 0 orders)
;= {["Wile Coyote" "Anvil"] 900,
;=  ["Elmer Fudd" "Anvil"] 300,
;=  ["Wile Coyote" "Hole"] 1000,
;=  ["Elmer Fudd" "Shells"] 100,
;=  ["Elmer Fudd" "Shotgun"] 800,
;=  ["Wile Coyote" "Dynamite"] 5000,
;=  ["Wile Coyote" "Clock"] 300}
```

Не совсем то, что ожидалось – мы не получили ассоциативный массив ассоциативных массивов. Эта проблема уходит корнями в `reduce-by`, где предполагается, что ассоциативный массив имеет плоскую структуру. Чтобы устранить ее, можно «исправить» `reduce-by`, создав версию для обработки вложенных ассоциативных массивов, или преобразовать ассоциативный массив, полученный в результате.

Чтобы обеспечить поддержку вложенных ассоциативных массивов в `reduce-by`, достаточно лишь заменить вызовы `assoc` и неявные вызовы `get` (когда ассоциативный массив используется в роли функции) вызовами `assoc-in` и `get-in`:

```
(defn reduce-by-in
  [keys-fn f init coll]
  (reduce (fn [summaries x]
```

```
            (let [ks (keys-fn x)]
              (assoc-in summaries ks
                (f (get-in summaries ks init) x))))
    {} coll))
```

Теперь мы получим двухуровневую сводную информацию:

```
(reduce-by-in (juxt :customer :product)
  #(+ %1 (:total %2)) 0 orders)
;= {"Elmer Fudd" {"Anvil" 300,
;=                "Shells" 100,
;=                "Shotgun" 800},
;=  "Wile Coyote" {"Anvil" 900,
;=                 "Hole" 1000,
;=                 "Dynamite" 5000,
;=                 "Clock" 300}}
```

Второй вариант заключается в преобразовании полученного набора данных:

```
(def flat-breakup
  {["Wile Coyote" "Anvil"] 900,
   ["Elmer Fudd" "Anvil"] 300,
   ["Wile Coyote" "Hole"] 1000,
   ["Elmer Fudd" "Shells"] 100,
   ["Elmer Fudd" "Shotgun"] 800,
   ["Wile Coyote" "Dynamite"] 5000,
   ["Wile Coyote" "Clock"] 300})
```

...в требуемый ассоциативный массив ассоциативных массивов. Воспользуемся для этого все той же функцией `assoc-in`:

```
(reduce #(apply assoc-in %1 %2) {} flat-breakup)
;= {"Elmer Fudd" {"Shells" 100,
;=                "Anvil" 300,
;=                "Shotgun" 800},
;= "Wile Coyote" {"Hole" 1000,
;=                "Dynamite" 5000,
;=                "Clock" 300,
;=                "Anvil" 900}}
```

Каждое значение в последовательности, получаемой из ассоциативного массива `flat-breakup`, представляет собой элемент ассоциа-

тивного массива в виде `[["Wile Coyote" "Anvil"] 900]`. То есть, когда наша функция свертки вызовет `apply` для каждого элемента ассоциативного массива, в результате будет вызвана `assoc-in` – например, `(assoc-in {} ["Wile Coyote" "Anvil"] 900)` – которая на основе данных в каждом элементе определит структуру ассоциативного массива, который должен получиться в результате, и его самые вложенные значения.

Неизменяемость и сохранность[1]

Мы разобрались с основными особенностями коллекций в языке Clojure и познакомились со многими их абстракциями. Но нам осталось исследовать еще две характеристики, свойственные всем структурам данных в Clojure: *неизменяемость* и *сохранность*.

В главе 2 мы познакомились с понятием неизменяемости и узнали, как семантика неизменяемости значений может существенно упростить работу с ними. Однако у нас еще остались некоторые нерешенные проблемы. Например, взгляните на следующую операцию над числами:

```
(+ 1 2)
;= 3
```

Здесь число `3` является значением, совершенно независящим от аргументов оператора `+`. Вне всяких сомнений, операция сложения чисел не изменяет слагаемые. Такой подход резко отличается от того, как действуют структуры данных в языках, поощряющих свободное изменение значений, таких как Python:

```
>>> lst = []
>>> lst.append(0)
>>> lst
[0]
```

Метод `append` действительно изменяет значение `lst`. Это влечет за собой массу последствий и не всегда приятных, зато такие операции весьма *эффективны* и скорость их выполнения почти не зависит от размера изменяемой коллекции. С другой стороны, такой порядок работы может порождать проблемы:

[1] Persistence. – *Прим. перев.*

```
(def v (vec (range 1e6)))    ❶
;= #'user/v
(count v)
;= 1000000
(def v2 (conj v 1e6))        ❷
;= #'user/v2
(count v2)
;= 1000001
(count v)                    ❸
;= 1000000
```

❶ Создается новый вектор, содержащий все целые числа в диапазоне от 0 до 1e6: миллион элементов – вполне обычный размер коллекции.
❷ С помощью conj в конец вектора добавляется еще одно целое число, в результате получается вектор, содержащий 1000001 элементов.
❸ Как уже говорилось, все структуры данных в Clojure являются неизменяемыми, поэтому вектор v не изменился.

Вектор v2 в этом примере – это отдельная структура данных. Кто-то может сказать: «Уверен, что это весьма неэффективно, потому что похоже, что conj (и, возможно, другие операции над структурами данных в языке Clojure) создает полную копию модифицируемой коллекции!»

К счастью это не так.

Сохранность и совместное использование

Операции над неизменяемыми структурами данных в языке Clojure *в действительности* весьма эффективны. Зачастую они выполняются с той же скоростью, что и эквивалентные им операции в языке Java. Это объясняется *сохранностью* (persistence) структур данных, реализацией приема повторного использования внутренних структур, что позволяет уменьшить количество операций для представления измененных версий экземпляра коллекции и гарантировать одинаковую эффективность обслуживания всех версий коллекции.

Примечание. Семантика «сохранности». Говоря о «сохранности», мы не подразумеваем сериализацию и сохранение данных, объектов и других значений. Понятие, обсуждаемое здесь, было введено Крисом Окасаки (Chris Okasaki) в его книге «Purely Functional Data Structures», где он описывает приемы, обеспечивающие сохранность гарантий высокой эффективности неизменяемых структур в более поздних версиях этих коллекций, полученных в результате различных операций..

Книга «Purely Functional Data Structures» представляет собой фундаментальный труд в области функционального программирования, на основополагающих идеях которой основаны архитектуры Clojure и других современных функциональных языков. Те, кто заинтересован в углублении своих знаний в функциональном программировании, тесно связанном со структурами данных, обязательно должны прочесть эту книгу.

Для достижения сохранности без ущерба для производительности, структуры данных в языке Clojure реализуют *совместное использование структур* (structural sharing). То есть, они никогда не копируются полностью – заново создаются только части, затронутые изменениями, а неизменившиеся части сохраняются в первоначальном виде.

Визуализация сохранности: списки

Действие этого механизма проще всего продемонстрировать на примере операций со списками. Взгляните на следующий список:

```
(def a (list 1 2 3))
```

Помня, что списки в Clojure являются связанными списками, графически их можно представить, как показано на рис. 3.4.

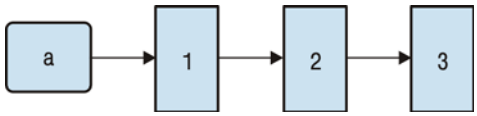

Рис. 3.4. Графическое представление связанного списка

Добавим с помощью `conj` новое значение в список. Не забывайте, что `conj` всегда добавляет значения в начало списка:

```
(def b (conj a 0))
;= #'user/b
b
;= (0 1 2 3)
```

Изобразить эту операцию можно, как показано на рис. 3.5.

Функция `conj` *создает* новый список с первым значением, равным `0`, но повторно использует список `a` целиком, в качестве хвоста нового списка. Эффективность этой операции очевидна:

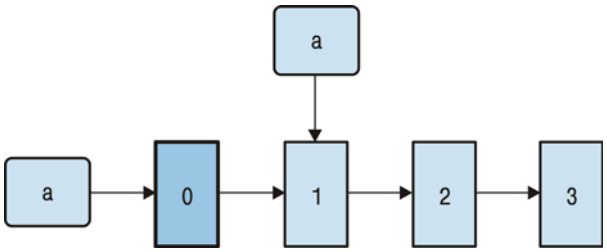

Рис. 3.5. Добавление нового элемента в список

- она не связана с копированием данных;
- оригинальный список остается нетронутым, он доступен и может использоваться как основа для создания других списков;
- новый список `b` совместно использует структуру `a` с единственным дополнительным значением.

Как уже говорилось выше, функция `conj` гарантирует постоянное время выполнения: добавление значения в коллекцию из трех элементов будет занимать то же время, что и добавление значения в коллекцию из миллиона элементов, и это утверждение верно в случае со списками. Таким образом, прежний список *сохраняется* в более поздней его версии – в списке `b`.

А что можно сказать о других операциях? Списки в языке Clojure не поддерживают произвольный доступ (как и любые другие связанные списки), по этой причине, например, время доступа к значениям внутри списка с помощью функции `nth` прямо пропорционально удаленности элемента от начала списка. Однако операция удаления первого элемента в списке выполняется весьма эффективно:

```
(def c (rest a))
;= #'user/c
c
;= (2 3)
```

Функция `rest` выполняет операцию со списком тоже за постоянное время, благодаря структуре данных, обеспечивающей возможность совместного использования в разных ее версиях. Функция `rest`[1] вернет хвост этого списка за то же время, независимо от размера исходного списка, и результат будет обеспечивать те же самые гарантии производительности для последующих операций.

[1] Здесь с тем же успехом можно было бы использовать функцию `pop`.

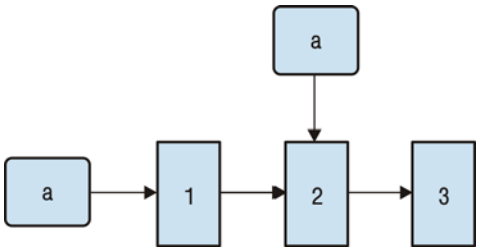

Рис. 3.6. Удаление первого элемента из списка

Визуализация сохранности: ассоциативные массивы (векторы и множества)

Итак, мы выяснили, что операции со связанными списками могут быть достаточно эффективными. А что можно сказать о структурах данных, наиболее часто используемых в программах на языке Clojure? Ассоциативные массивы, векторы и множества – все они реализуют ту же самую стратегию[1], даже при том, что их внутреннее устройство существенно сложнее, чем устройство списков.

Для дальнейшей демонстрации создадим такой ассоциативный массив (рис. 3.7):

```
(def a {:a 5 :b 6 :c 7 :d 8})
```

Ассоциативные массивы в языке Clojure реализованы в виде деревьев, где значения хранятся в листьях дерева.

Добавим новое значение в ассоциативный массив (рис. 3.8):

```
(def b (assoc a :c 0))
;= #'user/b
b
;= {:a 5, :c 0, :b 6, :d 8}
```

[1] Внутреннее устройство этих структур имеет множество мелких отличий, но ими можно пренебречь, обсуждая семантику сохранности в целом. Однако имейте в виду, что описание структуры ассоциативного массива, приводится в этом разделе исключительно в целях обсуждения семантики и не является точным отражением фактической реализации.

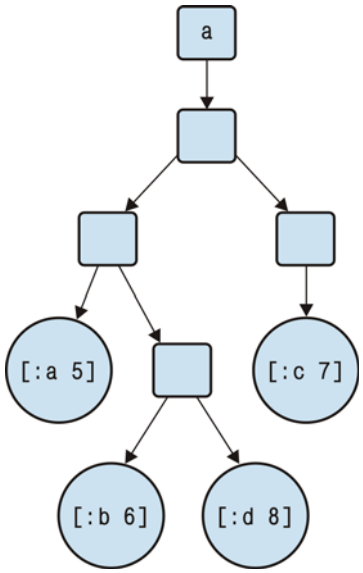

Рис. 3.7. Внутренняя организация ассоциативного массива `{:a 5 :b 6 :c 7 :d 8}`

Как и в примере, где новое значение добавлялось в список с помощью `conj`, операция `assoc` разделяет большую часть дерева, использующее структуру прежней версии ассоциативного массива. Повторно было использовано целое поддерево, содержащее элементы `:a`, `:b` и `:d` — в новое дерево добавился только один изменившийся лист, а остальные просто были присоединены к корню нового дерева.

Удаление элемента из ассоциативного массива дает тот же самый результат, если говорить в терминах совместного использования структуры (рис. 3.9):

```
(def c (dissoc a :d))
;= #'user/c
c
;= {:a 5, :c 7, :b 6}
```

И снова, большая часть структуры дерева не подверглась изменениям, и новое дерево совместно использует большую часть его структуры.

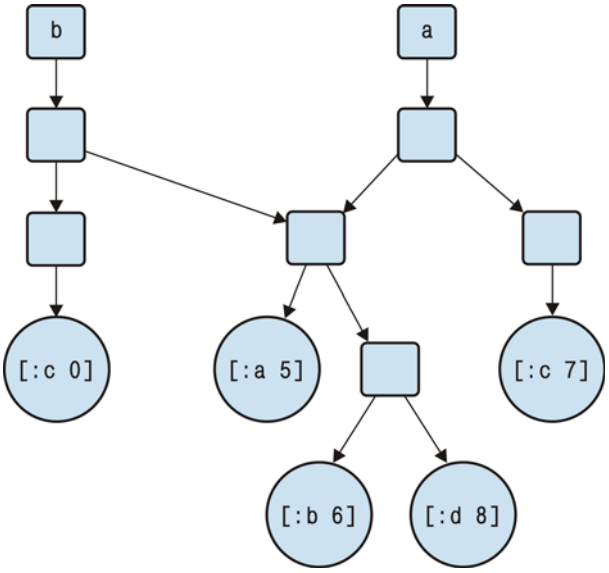

Рис. 3.8. Результат выполнения операции `(assoc a :c 0)`

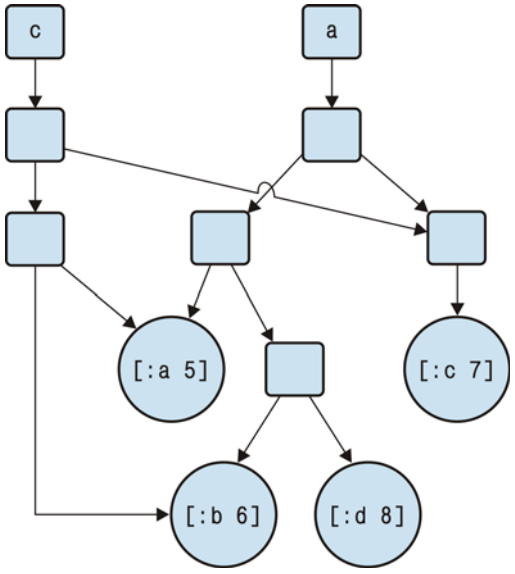

Рис. 3.9. Результат выполнения операции `(dissoc a :d)`

Лес деревьев

Чтобы обеспечить семантику сохранности, почти все структуры данных в языке Clojure реализованы в виде деревьев, включая ассоциативные массивы и множества, сортированные ассоциативные массивы и множества, а также векторы. Однако в каждом конкретном случае используются разные по структуре деревья: для представления ассоциативных массивов используется неизменяемый вариант отображения хеша в дерево массивов (hash array mapped trie)[1], для представления векторов используется другой вариант, называемый отображением массива в дерево хешей (array mapped hash trie)[2], множества основаны на ассоциативных массивах, а для представления сортированных множеств и сортированных ассоциативных массивов используются неизменяемые красно-черные деревья, что гарантирует высокую эффективность сортировки.

Во всех этих случаях используются реализации деревьев, *обеспечивающие* высокую скорость операций (таких как добавление в конец вектора, включение в множество, определение новой ассоциации в ассоциативном массиве и выполнение поиска в коллекциях любых типов). В терминах «большого О» эффективность этих операций характеризуется либо величиной $O(\log_{32} n)$ (для несортированных коллекций), либо $O(\log_2 n)$ (для сортированных коллекций), где n – число значений в коллекции. И что в результате? Операции с неизменяемыми структурами данных выполняются также быстро или почти так же быстро, как аналогичные операции с изменяемыми коллекциями, обеспечивая все выгоды семантики неизменяемости, о которых рассказывалось выше.

Интересно отметить, что многие системы заимствовали и применяют ту же стратегию использования неизменяемых и хранимых деревьев. Помимо Clojure в качестве примеров можно назвать Git и CouchDB. Краткий сравнительный обзор реализаций можно найти по адресу: http://eclipsesource.com/blogs/2009/12/13/persistent-trees-in-git-clojure-and-couchdb-data-structure-convergence.

Очевидные преимущества

Знакомство с деталями реализации структур данных в языке Clojure интересно уже само по себе, но кроме удовлетворения любопытства, мы получаем еще ряд более осязаемых преимуществ, которые

---

[1] Обзор реализации в виде класса `PersistentHashMap` можно найти в статье: http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice.

[2] Обзор реализации в виде класса `PersistentVector` можно найти в статье: http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation.

дают неизменяемость и сохранность. В главе 2 уже демонстрировалось, как применение неизменяемых значений может упростить программный код за счет устранения целых категорий ошибок и непредвиденных последствий, возникающих при использовании изменяемых структур данных. Но это далеко не все.

Поддержка конкуренции. Ссылочные типы в языке Clojure – описываемые в разделе «Ссылочные типы», в главе 4 – обязаны своим существованием эффективным и неизменяемым значениям, хранящимся в них. Поскольку эти ссылочные типы определяют семантику изменения некоторой идентифицируемой части данных с течением времени, было бы бессмысленно хранить в них изменяемые структуры данных. Любой другой код в любой момент смог бы изменить такие изменяемые коллекции и тем самым свести на нет все преимущества ссылочных типов и вновь ввергнуть вас в пучину проблем, связанных с синхронизацией одновременного доступа к данным.

Бесплатная поддержка версий. Одним из требований во многих системах является поддержка нескольких версий одних и тех же данных. Реализовать такую поддержку может оказаться достаточно сложно при использовании изменяемых структур данных. Как организовать копирование данных, чтобы последующие модификации не затронули их? Как наиболее корректно возвращать их в некоторое предыдущее состояние?

Неизменяемые коллекции позволяют легко решить эту проблему:

```
(def version1 {:name "Chas" :info {:age 31}})
;= #'user/version1
(def version2 (update-in version1 [:info :age] + 3))    ❶
;= #'user/version2
version1
;= {:info {:age 31}, :name "Chas"}
version2
;= {:info {:age 34}, :name "Chas"}
```

❶ `update-in` обновляет значение (определяемое вектором аргументов), находящееся в ассоциативной структуре (возможно вложенной), применяя указанную функцию (в данном случае – это `+`) с произвольными дополнительными аргументами (в данном случае – это `3`).

Каждая операция в Clojure, «изменяющая» коллекцию, оставляет эту коллекцию в исходном состоянии и возвращает новую ее вер-

сию. Каждая версия может использоваться, изменяться, и сохраняться, в соответствии с логикой приложения, но вам никогда не придется заботиться о поддержке версий данных, потому что она естественным образом вытекает из реализации коллекций.

Переходные структуры данных

Переходные (transient) коллекции – это зеркальное отражение сохраняемых коллекций: сохраняемые коллекции гарантируют целостность предыдущих версий значения, а переходные – нет. После изменения, любая ссылка на прежнюю версию переходной коллекции становится ненадежной – она может ссылаться на допустимое значение, на новое значение или на «мусор».

Следующая формулировка может показаться странной, но она появилось как вариация на тему старинного выражения: *если структура данных изменяется, и никто не видит как это происходит, действительно ли это кому-то может навредить?*

Переходные коллекции являются изменяемыми, резко выделяясь на общем фоне всех остальных структур данных в Clojure:

```
(def x (transient []))      ❶
;= #'user/x
(def y (conj! x 1))         ❷
;= #'user/y
(count y)                   ❸
;= 1
(count x)                   ❹
;= 1
```

❶ На основе неизменяемого вектора создается переходный вектор.
❷ С помощью функции `conj!` (аналог `conj`, но выполняющая операции с переходными коллекциями) в переходный вектор добавляется единственное значение...
❸ ...и возвращается новая ссылка на переходный вектор, содержащий единственное значение. Однако...
❹ ...ссылка на прежний переходный вектор отражает эти изменения[1]!

[1] Как уже отмечалось в начале этого раздела, ссылки на прежние версии переходных коллекций являются ненадежными – этот пример лишь иллюстрирует изменчивость переходных коллекций и что их семантика противоречит семантике неизменяемых коллекций.

С учетом сказанного выше возникает вопрос: где могут пригодиться переходные структуры данных? Неизменяемые структуры данных в Clojure обладают множеством *весьма* привлекательных характеристик и вдобавок ко всему они *очень* эффективны. Однако есть некоторые обстоятельства, которые невозможно игнорировать: основная проблема заключается в необходимости размещения новых значений при каждом обновлении, что сопряжено с некоторыми накладными расходами. Накладные расходы становятся особенно заметными, когда изменения носят массовый характер, например, при добавлении в коллекции сотен и тысяч значений.

Переходные структуры данных предусмотрены лишь для оптимизации подобных случаев. Иногда переходные коллекции могут уменьшить или вообще устранить эффект накопления накладных расходов на размещение объектов, снижая время, необходимое на сборку мусора, и повышая скорость выполнения операций с коллекциями. В частности, они легко позволяют ускорить обработку объектов в продолжительных циклах[1].

Вам интересно, какие ситуации имеются в виду? Хорошо, возьмем для примера базовую функцию `into`, с которой мы уже встречались выше. Она принимает коллекцию и некоторую последовательность значений, и добавляет последнюю в первую:

```
(into #{} (range 5))
;= #{0 1 2 3 4}
```

Попробуем реализовать свою версию. Ниже приводится первый вариант, не самый лучший:

```
(defn naive-into
  [coll source]
  (reduce conj coll source))

(= (into #{} (range 500))
   (naive-into #{} (range 500)))
;= true
```

[1] Столкнувшись с проблемами низкой производительности, размещение новых объектов – это первое, на что следует обратить внимание при профилировании программ на языке Clojure.

Отлично, теперь у нас есть собственная реализация `into` и она работает с любыми неизменяемыми коллекциями. А теперь сравним производительность нашей версии и стандартной:

```
(time (do (into #{} (range 1e6))    ❶
          nil))
; "Elapsed time: 1756.696 msecs"
(time (do (naive-into #{} (range 1e6))
          nil))
; "Elapsed time: 3394.684 msecs"
```

❶ Мы не будем выводить миллион чисел в REPL, поэтому просто вернем `nil` из формы `do`.

Ох! Функция `naive-into` оказалась почти в два раза медленнее! Причина в том, что при наличии возможности `into` использует переходную коллекцию[1], то есть, когда в первом аргументе передается вектор, несортированный ассоциативный массив или несортированное множество (единственные типы коллекций, для которых можно создавать переходные варианты).

Попробуем сделать то же самое:

```
(defn faster-into
  [coll source]
  (persistent! (reduce conj! (transient coll) source)))
```

Перед началом свертки мы преобразуем исходную коллекцию в переходную. На каждом шаге операции свертки вызывается функция `conj!` – переходный аналог `conj` – добавляющая следующее значение из `source` в переходную коллекцию. По окончании `reduce` возвращает новую переходную коллекцию, которая преобразуется в неизменяемую вызовом функции `persistent!`. И что в результате?

```
(time (do (faster-into #{} (range 1e6))
          nil))
; "Elapsed time: 1639.156 msecs"
```

Производительность стала сравнима с производительностью `into`. А теперь отступим на шаг назад: практически всегда «быстрее»

---

[1] Другие базовые функции Clojure так же используют переходные коллекции за кулисами, когда это возможно, включая `frequencies` и `group-by`.

означает «лучше», но не потеряли ли мы преимущества, которые нам дает неизменяемость? Если говорить коротко – нет. Да, в процессе свертки, внутри функции `faster-into`, операции выполняются с изменяемой коллекцией, но *эта переходная коллекция не покидает область видимости функции, использующей ее*! Это означает, что `faster-into` имеет точно такую же семантику, что и `naive-into` – она получает и возвращает неизменяемую коллекцию. То есть пользователи получают все преимущества семантики неизменяемых коллекций и высокую скорость обработки изменяемых данных.

Внимание. Не забывайте, что переходные аналоги существуют только для векторов, несортированных ассоциативных массивов и несортированных множеств. Поэтому `faster-into` потерпит неудачу, если попытаться передать ей в первом аргументе, например, сортированное множество. К сожалению, не существует стандартного предиката, позволяющего определить наличие переходного варианта для указанного типа коллекций. Чтобы выяснить это, необходимо проверить, является ли данная коллекция экземпляром интерфейса `clojure.lang.IEditableCollection`, указывающего, что для коллекции можно создать переходный вариант. Предикат, например `transient-capable?`, выполняющий такую проверку, мог бы выглядеть так:

```
(defn transient-capable?
  "Возвращает true, если для данной коллекции можно получить
   переходный вариант. То есть, проверяет - преуспеет ли
   `(transient coll)`."
  [coll]
  (instance? clojure.lang.IEditableCollection coll))
```

Теперь, когда мы познакомились с преимуществами переходных структур данных и определили границы области, где они могут использоваться, рассмотрим некоторые особенности их механизма.

В соответствии с названием, неизменяемые коллекции, служащие основой для создания переходных коллекций, не изменяются:

```
(def v [1 2])
;= #'user/v
(def tv (transient v))
;= #'user/tv
(conj v 3)
;= [1 2 3]
```

С другой стороны, превращение переходной коллекции в неизменяемую, с помощью persistent!, делает исходную переходную коллекцию непригодной[1]:

```
(persistent! tv)
;= [1 2]
(get tv 0)
;= #<IllegalAccessError java.lang.IllegalAccessError:
;= Transient used after persistent! call>
```

Обе функции, transient и persistent! выполняют свою работу за постоянное время.

Переходные коллекции поддерживают множество функций доступа, аналогичных тем, что имеются для их неизменяемых аналогов, но не все:

```
(nth (transient [1 2]) 1)
;= 2
(get (transient {:a 1 :b 2}) :a)
;= 1
((transient {:a 1 :b 2}) :a)      ❶
;= 1
((transient [1 2]) 1)
;= 2
(find (transient {:a 1 :b 2}) :a)
;= #<CompilerException java.lang.ClassCastException:
;= clojure.lang.PersistentArrayMap$TransientArrayMap
;= cannot be cast to java.util.Map (NO_SOURCE_FILE:0)>
```

❶ Переходные коллекции тоже являются функциями!

Одним из известных исключений является функция seq, не поддерживающая переходные коллекции. Это объясняется тем, что последовательность может пережить свой источник, и переходные коллекции не дают гарантий, которые обеспечивают последовательности, как неизменяемые коллекции.

Кроме того, ни одна из функций, используемых для модификации неизменяемых коллекций, не может применяться к переходным аналогам. Переходные коллекции поддерживают свое, отдельное мно-

[1] Восклицательный знак в конце имени persistent! указывает, что это – деструктивная операция.

жество функций, выполняющих те же операции, что и их аналоги: `conj!`, `assoc!`, `dissoc!`, `disj!`, and pop!.

Имена всех функций для работы с переходными коллекциями оканчиваются восклицательным знаком, указывающим на особенности их поведения в отношении коллекции, передаваемой в первом аргументе. После применения любой из этих функций к переходной коллекции, *эта коллекция никогда не должна затрагиваться снова* – даже только для чтения. Так же, как всегда следует использовать результат функций `conj`, `assoc`, `disj` и других, чтобы получить эффект от их применения, вы всегда должны использовать результаты их переходных аналогов. То есть, нельзя продолжать оперировать переходной коллекцией на месте, иначе результаты могут оказаться неожиданными:

```
(let [tm (transient {})]
  (doseq [x (range 100)]
    (assoc! tm x 0))          ❶
  (persistent! tm))
;= {0 0, 1 0, 2 0, 3 0, 4 0, 5 0, 6 0, 7 0}
```

❶ Даже при том, что с помощью `assoc!` мы поместили в переходный ассоциативный массив 100 элементов, практически все они были потеряны, потому что мы не использовали результат, возвращаемый `assoc!`.

Переходные структуры данных имеют то же предназначение, что и их неизменяемые аналоги, с дополнительным ограничением – предыдущие версии переходных структур данных не должны использоваться повторно. Самый простой и надежный способ работы с переходными структурами данных состоит в том, чтобы писать обычный программный код на языке Clojure, добавляя необходимые вызовы `transient` и `persistent!`, и вставлять восклицательные знаки в конец имен функций, возвращающих измененные переходные коллекции. При этом необходимо соблюдать линейность операций с переходными коллекциями (то есть, никогда не использовать прежнюю версию переходной коллекции после ее изменения). Как было показано на примере реализации `faster-into`, свертка является, пожалуй, самой простой операцией преобразования, используемой с переходными структурами данных.

Переходные структуры данных предназначены исключительно для оптимизации, поэтому они должны использоваться с осторожностью и только локально, обычно внутри единственной функции

(или в пределах библиотеки, в группе взаимосвязанных приватных функций). Некоторые из этих правил носят принудительный характер, например, ограничитель конкуренции, встроенный в переходные структуры данных, гарантирует, что изменить ее сможет только поток выполнения, создавший данную переходную коллекцию. Подробнее о конкуренции будет рассказываться в главе 4, а пока достаточно отметить, что функция `future` в следующем фрагменте обеспечит вызов `get` в другом потоке выполнения, отличном от того, который создает переходный ассоциативный массив и связывает его с локальной переменной `t`:

```
(let [t (transient {})]
  @(future (get t :a)))
;= #<IllegalAccessError java.lang.IllegalAccessError:
;= Transient used by non-owner thread>
```

Переходные структуры данных дают возможность оптимизировать производительность, но за это приходится платить необходимостью реструктуризации кода из-за имеющихся ограничений и отсутствия некоторых стандартных функций для работы с ними. Как и при использовании любых приемов оптимизации, необходимо анализировать – действительно ли данная стратегия является лучшей и обеспечивает ли она необходимую эффективность.

Переходные структуры данных не поддерживают вложенность. Функция `persistent!` не выполняет обход иерархии вложенных переходных структур данных, поэтому вызов `persistent!` для ссылки верхнего уровня не окажет влияния на вложенные коллекции:

```
(persistent! (transient [(transient {})]))
;= [#<TransientArrayMap clojure.lang.
;=     PersistentArrayMap$TransientArrayMap@b57b39f>]
```

В любом случае из-за своей изменчивости переходные коллекции не поддерживают семантику значений и не могут интерпретироваться, как показано ниже:

```
(= (transient [1 2]) (transient [1 2]))
;= false
```

Это еще раз подводит нас к мысли, что переходные структуры данных со своей изменяемостью не должны смешиваться с кол-

лекциями, обладающими классической семантикой. Они должны использоваться только для локальной оптимизации, скрытой от окружающего кода, как в известной поговорке про звук падающего дерева в лесу, который некому услышать.

Метаданные

Метаданные – это данные, описывающие данные. Метаданные имеют множество других имен и принимают самые разные формы в других языках:

- объявления типов и модификаторы доступа (такие как `private`, `protected` и другие) являются метаданными, описывающими значения, переменные и функции, с которыми они ассоциированы;
- аннотации в Java являются метаданными, описывающими классы, методы, аргументы методов и так далее.

Метаданные в языке Clojure используются с теми же целями[1], однако они представляют собой нечто большее, чем некоторое универсальное средство для применения к данным в приложении.

Метаданные можно присоединять к любым структурам данных Clojure, последовательностям, записям, символам или ссылочным типам, и они всегда имеют форму ассоциативного массива. Механизм чтения имеет удобный синтаксис декларативного присоединения метаданных к литералам значений:

```
(def a ^{:created (System/currentTimeMillis)}
        [1 2 3])
;= #'user/a
(meta a)
;= {:created 1322065198169}
```

Для удобства метаданные, содержащие только слоты с ключами, являющимися ключевыми словами, и логическим значением `true`, могут записываться в сокращенной форме и аддитивно применяться к значению, которое будет прочитано следующим:

---

[1] Использование метаданных для передачи информации о типах компилятору Clojure описывается в разделе «Указание типов для производительности», в главе 9; об определении политики доступа к переменным и семантики конкуренции рассказывается в разделе «Переменные», в главе 4; а применение аннотаций Java к типам Clojure освещается в разделе «Аннотации», в главе 9.

```
(meta ^:private [1 2 3])
;= {:private true}
(meta ^:private ^:dynamic [1 2 3])
;= {:dynamic true, :private true}
```

Обновлять метаданные, приложенные к данному значению, можно с помощью функций `with-meta` и `vary-meta`:

```
(def b (with-meta a (assoc (meta a)
                      :modified (System/currentTimeMillis))))
;= #'user/b
(meta b)
;= {:modified 1322065210115, :created 1322065198169}
(def b (vary-meta a assoc :modified (System/currentTimeMillis)))
;= #'user/b
(meta b)
;= {:modified 1322065229972, :created 1322065198169}
```

Функция `with-meta` замещает метаданные для значения, а `vary-meta` обновляет уже имеющийся ассоциативный массив с метаданными, вызывая указанную функцию (`assoc` в примере выше) с дополнительными аргументами.

Не забывайте, что метаданные – это данные, *описывающие* другие данные. Иными словами, изменение метаданных значения не влияет на то, как будет выводиться значение или как будет определяться его равенство (или неравенство) с другими значениями:

```
(= a b)
;= true
a
;= [1 2 3]
b
;= [1 2 3]
(= ^{:a 5} 'any-value
^{:b 5} 'any-value)
;= true
```

Конечно, значения с метаданными являются такими же неизменяемыми, как значения без метаданных; то есть операции, «модифицирующие» структуры данных, возвращают новые структуры данных, сохраняя оригинальные метаданные:

```
(meta (conj a 500))
;= {:created 1319481540825}
```

Это намного более полезно в сравнении с обработкой метаданных уровня приложения в других языках, где такая информация хранится отдельно от «действительных» данных, чтобы не оказывать нежелательного влияния на операции сравнения и избавиться от необходимости переносить ее в обновленные значения и агрегаты. Выше уже демонстрировалось, что в метаданных может храниться такая информация, как время создания и изменения; представьте так же, что можно хранить информацию о происхождении данных, загружаемых из разных источников, такую как сведения о базе данных и кеше. Значения, представляющие данные из таких источников, можно снабжать разными метаданными, не влияя на операции с ними, и затем использовать их, чтобы определить, как обрабатывать эти значения: например, куда направлять запросы на обновление значения, загруженного из базы данных или из кеша.

Включаем коллекции Clojure в работу

Почему структуры данных так важны? В книге «The Mythical Man-Month»[1], Фредерик Брукс (Frederick Brooks) говорит:

> Покажите мне блок-схемы, не показывая таблиц, и я останусь в заблуждении. Покажите мне ваши таблицы, и блок-схемы, скорей всего, не понадобятся: они будут очевидны.

Спустя двадцать лет Эрик Реймонд (Eric Raymond) перефразировал это высказывание:

> Покажите мне код, не показывая структур данных, и я останусь в заблуждении. Покажите мне ваши структуры данных, и код, скорей всего, не понадобится: он будет очевиден.

Структура и модель данных определяют форму программного кода, следовательно невозможно написать хороший функциональный код на языке Clojure, следуя старым привычкам и моделируя все подряд на основе массивов, выделенных объектов и иногда ассоциативных массивов.

---

[1] Фредерик Брукс, «Мифический человеко-месяц, или как создаются программные системы», Символ-Плюс, 2007, ISBN: 5-93286-005-7. – *Прим. перев.*

При моделировании данных в языке Clojure, в центре внимания должны быть значения (и в частности – составные значения), естественные идентификаторы, а также множества и ассоциативные массивы. Ход мыслей при этом должен напоминать ход мыслей при реляционном моделировании[1]. Однако само понятие «значение» тесно связано с контекстом: составные значения могут иметь иерархическую организацию, поэтому разработчик должен определить степень детализации отношений, которую он будет поддерживать.

Идентификаторы и циклы

Естественные идентификаторы являются полной противоположностью синтетическим, или суррогатным идентификаторам (последние обычно являются строками или искусственно сгенерированными числами), и напоминают последовательности, автоинкрементные поля в базах данных или ссылки объектов на самих себя в большинстве объектно-ориентированных языков. То есть, синтетическим идентификаторам по определению свойственна излишняя сложность.

Без этих посредников отпадает необходимость обеспечивать каноническое отображение *компонентов* в их *идентификаторы* (ID): либо «компоненты» уникальны и, как значения, в состоянии идентифицировать себя, либо нет. Как дополнительное преимущество, при отказе от синтетических идентификаторов отпадает необходимость взаимодействий между разными потоками выполнения и даже процессами, с целью присвоить уникальные идентификаторы фрагментам данных, потому что естественные идентификаторы можно определить на основе самих данных.

Споры о преимуществах синтетических ключей перед естественными ведутся уже достаточно давно, и очень часто вспыхивают с новой силой, когда дело доходит до проектирования схем баз данных. В пользу синтетических идентификаторов обычно приводятся два основных аргумента: данные живут дольше процесса (и правил, которые они отражают), создавшего их, и эти данные тяжело поддаются реорганизации. Как следствие, синтетические идентификаторы мало пригодны для использования в прикладных вычислениях.

Типичным примером ненужности синтетических идентификаторов являются различные состояния, генерируемые парсерами или

---

[1] Пространство имен `clojure.set` является реализацией операторов реляционной алгебры (relational algebra).

механизмами регулярных выражений. Обычно каждое состояние парсера нумеруется чисто механически, однако в Clojure есть возможность конструировать сложные значения состояний, полностью описывающие себя семантически значимым способом. Благодаря этому вычисление переходных структур данных на основе состояний может быть выполнено с помощью простой функции, принимающей в виде аргумента только фактическое состояние. В отличие от распространенной практики, отпадает необходимость передавать числовое представление состояния вместе с данными, описывающими это состояние – достаточно одних лишь данных.

Различные состояния можно продолжать нумеровать по соображениям эффективности, но вы должны помнить, что:

1. Это необязательно.
2. Перенумерацию или переименование можно выполнить отдельно, явно отделив проблемы и сложности. То есть, при необходимости можно продолжать поддерживать разнообразные синтетические идентификаторы, часто требуемые при работе с данными, которыми владеют или которые используют другие взаимодействующие системы.

Единственным табу в общей картине являются циклы: единственный разумный способ реализовать их связан с использованием идентификаторов и косвенных ссылок. Аналогично, отсутствие циклов означает отсутствие обратных ссылок.

Циклы, значения и идентичность. Предположим, что имеется неизменяемое дерево узлов (например, узлов DOM), обладающих слотами `:parent-node` и `:children`. Что произойдет с этими слотами, если «обновить» (создать новую версию) некоторого узла? Если значения с ключами `:parent-node` и `:children` не изменялись, то переход по ссылкам в этих свойствах туда-обратно вернет вас... к старому, неизмененному значению узла, а не к новому! Весьма неприятная ситуация, вынуждающая переустанавливать ссылки в родительском и в дочерних узлах на измененный узел.

Более того, чтобы избежать ошибок, это изменение должно быть распространено по всем узлам в дереве. Для этой цели можно было бы использовать ссылочные типы, которые подробно будут обсуждаться в разделе «Состояние и идентичность», в главе 4. Но в результате этого дерево перестанет быть неизменяемым значением, а вокруг узлов придется наращивать специальную логику, обеспечивающую непротиворечивость изменяемых ссылок. Добро пожало-

вать обратно в мир необузданной изменчивости и непредвиденных сложностей!

В большинстве случаев циклы (они же обратные ссылки) встроены в структуры данных для навигации. Такие ситуации покрываются маршрутами (paths), или *зипперами* (zippers), о которых будет рассказываться в разделе «Навигация, изменение и зипперы», ниже. В остальных случаях для введения циклов вы вынуждены будете добавлять уровень косвенности.

Такой уровень косвенности может быть основан на простых ссылочных типах или идентификаторах (используемых для поиска в индексе, обычно реализованном в виде ассоциативного массива). Порой это может вызывать тяжелые чувства, но ломающиеся циклы будут вынуждать вас задаваться важным вопросом – какие части данных являются чистыми значениями, а какие заслуживают создания идентификаторов, и какими должны быть эти идентификаторы[1]? Эти вопросы и ответы на них позволят сэкономить время позднее, когда наступит момент выпустить данные за пределы приложения, отправляя их в сеть, сохраняя на диск, записывая в базу данных или экспортируя и посредством какой-либо службы.

Думайте иначе: от императивного к функциональному

Мы уже привели доводы против числовых идентификаторов и рекомендовали использовать вместо них последовательности. Однако, несмотря на то, что последовательности являются отличной абстракцией, они остаются линейными и сами по себе не помогут взглянуть на проблемы под другим углом и найти эффективное решение с применением структур данных языка Clojure. Вам необходимо полностью переключиться на программирование с применением значений и использовать их в полном объеме, чтобы извлечь максимум возможного из языка Clojure.

---

[1] Проблема определения идентичности системы и ее границ тесно связана с проблемами предметно-ориентированного проектирования (domain-driven design), которые исследуются в книге Эрика Эванса (Eric Evans) «Domain-Driven Design: Tackling Complexity in the Heart of Software» (Addison-Wesley Professional) (Эрик Эванс, «Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем», Вильямс, 2010, ISBN: 978-5-8459-1597-9. – *Прим. перев.*).

Вспоминаем классику: игра «Жизнь»

Игра «Жизнь», придуманная английским математиком Джоном Конвеем (John Conway) (https://ru.wikipedia.org/wiki/Жизнь_(игра)) является алгоритмом, для реализации которого, кажется, лучше всего подойдут массивы. Мы реализуем правила этой игры: сначала традиционным способом – в качестве игрового поля будет использоваться вектор векторов, каждый элемент которого будет иметь либо значение `:on`, либо `nil`, а затем в более идиоматичной для языка Clojure манере – без сложностей (и ограничений), присущих индексам.

```
(defn empty-board
  "Создает прямоугольное игровое поле с указанной шириной и высотой."
  [w h]
  (vec (repeat w (vec (repeat h nil)))))
```

Теперь у нас появилась возможность создавать пустое игровое поле и нам необходимо реализовать добавление живых клеток в него:

```
(defn populate
  "Включает значение :on в ячейках, определяемых координатами [y, x]."
  [board living-cells]
  (reduce (fn [board coordinates]
            (assoc-in board coordinates :on))
          board
          living-cells))

(def glider (populate (empty-board 6 6) #{[2 0] [2 1] [2 2] [1 2] [0 1]}))

(pprint glider)
; [[nil :on nil nil nil nil]
;  [nil nil :on nil nil nil]
;  [:on :on :on nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]]
```

Теперь самое основное: функция `indexed-step`, принимающая состояние игрового поля и возвращающая следующее его состояние в соответствии с правилами игры:

Пример 3.6. Реализация вспомогательных функций для indexed-step

```
(defn neighbours
  [[x y]]
```

```
  (for [dx [-1 0 1] dy [-1 0 1] :when (not= 0 dx dy)]
    [(+ dx x) (+ dy y)]))

(defn count-neighbours
  [board loc]
  (count (filter #(get-in board %) (neighbours loc))))     ❶

(defn indexed-step
  "Возвращает следующее состояние игрового поля, используя индексы для
   определения координат ячеек, соседних с живыми клетками."
  [board]
  (let [w (count board)
        h (count (first board))]
    (loop [new-board board x 0 y 0]
      (cond
        (>= x w) new-board
        (>= y h) (recur new-board (inc x) 0)
        :else
          (let [new-liveness
                   (case (count-neighbours board [x y])
                     2 (get-in board [x y])
                     3 :on
                     nil)]
            (recur (assoc-in new-board [x y] new-liveness) x (inc y)))))))
```

❶ Обратите внимание: из-за того, что `count-neighbours` использует `get-in` — реализованную поверх `get`, которая, как мы знаем, возвращает `nil` для неизвестных индексов, она не генерирует ошибки при выходе за пределы игрового поля.

Посмотрим, как все это работает:

```
(-> (iterate indexed-step glider) (nth 8) pprint)
; [[nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil :on nil nil]
;  [nil nil nil nil :on nil]
;  [nil nil :on :on :on nil]
;  [nil nil nil nil nil nil]]
```

Теперь у нас есть действующая реализация игры «Жизнь» с колонией клеток!

Посмотрим, как можно переделать это решение, чтобы избавиться от индексов. Для начала уберем итерации, выполняемые вручную. Любой вызов `loop` можно заменить вызовом `reduce` для диапазона:

```
(defn indexed-step2
  [board]
  (let [w (count board)
        h (count (first board))]
    (reduce
      (fn [new-board x]
        (reduce
          (fn [new-board y]
            (let [new-liveness
                     (case (count-neighbours board [x y])
                       2 (get-in board [x y])
                       3 :on
                       nil)]
              (assoc-in new-board [x y] new-liveness)))
          new-board (range h)))
      board (range w))))
```

Вложенные операции свертки всегда можно сократить:

```
(defn indexed-step3
  [board]
  (let [w (count board)
        h (count (first board))]
    (reduce
      (fn [new-board [x y]]
        (let [new-liveness
                 (case (count-neighbours board [x y])
                   2 (get-in board [x y])
                   3 :on
                   nil)]
          (assoc-in new-board [x y] new-liveness)))
      board (for [x (range h) y (range w)] [x y]))))
```

Мы получили версию без цикла `loop`, в которой, однако, все еще используются индексы.

Как уже говорилось, индексы можно заменить последовательностями, но наши функции `count-neighbours` и `neighbours` тесно связаны с индексами, используя их для определения соседних ячеек и доступа к ним. Как можно выразить понятие «соседа» с помощью последовательности и без использования индексов?

При использовании одномерного массива сделать это очень просто, достаточно воспользоваться функцией `partition`:

```
(partition 3 1 (range 5))
;= ((0 1 2) (1 2 3) (2 3 4))
```

Результат, возвращаемый функцией `partition`, здесь можно интерпретировать как последовательность элементов 1, 2 и 3 с их соседями. Единственная проблема состоит в том, что этот код создает «окна» только для элементов, имеющих соседей с обеих сторон: элементы 0 и 4 с их соседями отсутствуют! Эту проблему можно исправить, дополнив исходную коллекцию:

```
(partition 3 1 (concat [nil] (range 5) [nil]))
;= ((nil 0 1) (0 1 2) (1 2 3) (2 3 4) (3 4 nil))
```

Оформим эту операцию в виде функции `window`:

```
(defn window
  "Возвращает ленивую последовательность окон с тремя элементами в каждом,
   центрами в которых является элемент coll."
  [coll]
  (partition 3 1 (concat [nil] coll [nil])))
```

Но как теперь перейти к двум измерениям? Вся хитрость в том, что когда мы будем применять функцию `window` к коллекции из `n` строк, мы будем получать `n` троек из трех строк, а каждая такая тройка (длиной `m`) может быть преобразована в последовательность из `m` троек. Формально эта операция называется *транспонированием* (transposition). Повторно применив `window` к такой последовательности, мы получим последовательность троек из троек и сможем создать окно 3 на 3 вокруг каждого элемента.

Взгляните на код:

```
(defn cell-block
  "Создает последовательность окон 3x3 на основе тройки из трех
   последовательностей."
  [[left mid right]]
  (window (map vector
           (or left (repeat nil)) mid (or right (repeat nil)))))
```

Две формы `or` должны заменить дополнительные значения `nil`, сгенерированные функцией `window` повторяющимися последовательностями из `nil`, потому что `map` остановится, как только один из ее

аргументов окажется пустым[1]. Этот код можно упростить, добавив в функцию window необязательный аргумент pad:

```
(defn window
  "Возвращает ленивую последовательность окон с тремя элементами в каждом,
   центрами в которых является элемент coll, и дополненную значением
   pad или nil, если необходимо."
  ([coll] (window nil coll))
  ([pad coll]
    (partition 3 1 (concat [pad] coll [pad]))))

(defn cell-block
  "Создает последовательности окон 3x3 из троек последовательностей по 3
   элемента в каждой."
  [[left mid right]]
  (window (map vector left mid right)))
```

Нам необходимо определить, является ли ячейка в центре блока живой клеткой. Эту операцию так же желательно оформить в виде отдельной функции, но на сей раз воспользуемся механизмом деструктуризации, чтобы сжато разделить блок ячеек на составляющие:

```
(defn liveness
  "Возвращает признак наличия живой клетки (nil или :on) в центральной
   ячейке для выполнения следующего шага."
  [block]
  (let [[_ [_ center _] _] block]
    (case (- (count (filter #{:on} (apply concat block)))
             (if (= :on center) 1 0))
      2 center
      3 :on
      nil)))
```

Теперь можно переделать функцию indexed-step, задействовав в ней вспомогательные функции, не зависящие от индексов:

```
(defn- step-row
  "Возвращает следующее состояние центра строки."
  [rows-triple]
```

[1] Иными словами, результат функции map имеет длину, равную наиболее короткому из ее аргументов; то есть, если один из аргументов будет иметь значение nil или окажется пустым, map вернет пустую последовательность.

```
  (vec (map liveness (cell-block rows-triple))))

(defn index-free-step
  "Возвращает следующее состояние игрового поля."
  [board]
  (vec (map step-row (window (repeat nil) board))))
```

Даже при том, что `index-free-step` опирается на вспомогательную функцию, не зависящую от индексов, она эквивалентна функции `indexed-step`:

```
(= (nth (iterate indexed-step glider) 8)
   (nth (iterate index-free-step glider) 8))
;= true
```

Каждый шаг в проделанном нами пути достаточно прост, но весь путь от «императивного» решения к «последовательному» может показаться слишком заумным, пока вы не пройдете по нему пару раз.

Переход на следующий уровень. Проблема текущей реализации состоит в том, что она остается близкой по духу к *первоначальной реализации*. Однако можно попробовать найти более изящное решение. Для этого нужно глубоко вдохнуть, отступить на шаг назад, и детально исследовать правила игры «Жизнь».

На каждом шаге выполняются следующие переходы:

- любая живая клетка, имеющая по соседству менее двух живых клеток, погибает из-за малонаселенности;
- любая живая клетка, имеющая по соседству две или три живые клетки, продолжает жить в следующем поколении;
- любая живая клетка, имеющая по соседству более трех живых клеток, погибает из-за перенаселенности;
- в любой пустой ячейке, по соседству с которой имеется точно три живые клетки, появляется новая живая клетка в результате размножения.

В этих правилах не упоминаются ни строки, ни столбцы, ни индексы. В них говорится только о соседних ячейках и о наличии живых клеток в них. Точнее, в них говорится о ячейках с живыми клетками, о соседних ячейках и о пустых ячейках рядом с живыми клетками. Таким образом, двумя основными понятиями являются живые клетки и соседние ячейки, а понятие пустых ячеек (соседние пустые ячейки) выводится из понятий *соседние ячейки* и *живые клетки*.

Если придерживаться этих двух понятий, *единственным состоянием мира является множество живых клеток*. Чтобы сгенерировать каждое последующее состояние, достаточно сначала определить всех живых соседей ячейки и затем подсчитать, сколько раз появляется данная «соседняя ячейка» (количество появлений будет соответствовать количеству живых соседей).

Если попытаться переложить это на язык Clojure, в конечном итоге получится:

Пример 3.7. Изящная реализация игры «Жизнь»

```
(defn step
  "Возвращает следующее состояние мира"
  [cells]
  (set (for [[loc n] (frequencies (mapcat neighbours cells))
             :when (or (= n 3) (and (= n 2) (cells loc)))]
         loc)))
```

Это все! Нам потребовалась единственная вспомогательная функция `neighbours`; здесь не нужны индексы, игровое поле в виде вектора векторов, отсутствуют ограничения на размер игрового поля, и игровой мир имеет разреженное представление – явно представлены только координаты ячеек с живыми клетками[1].

Опробуем эту новую функцию `step` с нашим начальным игровым полем; Состояние мира больше не является игровым полем – оно хранит только координаты с живыми ячейками – но мы можем повторно использовать функцию `populate` для создания ограниченного игрового поля, которое проще визуализировать:

```
(->> (iterate step #{[2 0] [2 1] [2 2] [1 2] [0 1]})   ❶
  (drop 8)
  first
  (populate (empty-board 6 6))
  pprint)
; [[nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil :on nil nil]
;  [nil nil nil nil :on nil]
;  [nil nil :on :on :on nil]
;  [nil nil nil nil nil nil]]
```

[1] Наравне с языками обработки массивов, такими как APL или J, несмотря на фундаментальные различия между реализациями.

❶ Начиная с того же исходного расположения живых клеток, мы можем видеть, что колония перемещается на одну ячейку через каждые четыре шага.

Интересно отметить, что функция `step` в примере 3.7 использует функцию `neighbours`, реализованную для императивной функции `indexed-step` в примере 3.6. Однако, в контексте данного решения, `neighbours` имеет дело *не* с числовыми индексами в конкретной структуре данных, а с координатами: пары `[x y]` являются очевидными идентификаторами в отношении функции `step`, тогда как в исходной императивной функции `indexed-step` эти же пары определялись из индексов самой функцией `indexed-step`.

Таким образом, `neighbours` теперь единственная часть данного алгоритма, где требуется определять содержимое идентификаторов ячеек. Фактически она определяет топологию решетки. Немного изменив `neighbours`, этот код сможет поддерживать конечные решетки, тороидальные решетки, гексагональные решетки, N-мерные решетки и так далее, без изменения функции `step`. Уйдя от императивного мышления, мы получили более ясное разделение проблем и более удачное решение, очень близкое к универсальному.

Из функции `step` мы легко можем сделать нечто действительно универсальное: функцию высшего порядка, `stepper`, действующую как фабричную, производящую функции `step`.

```
(defn stepper
  "Возвращает функцию step для реализации клеточного автомата.
   neighbours принимает координаты и возвращает упорядоченную коллекцию
   координат. Функции survive? и birth? – это предикаты, проверяющие
   число живых соседей."
  [neighbours birth? survive?]
  (fn [cells]
    (set (for [[loc n] (frequencies (mapcat neighbours cells))
               :when (if (cells loc) (survive? n) (birth? n))]
           loc))))
```

Наша реализация функции `step` эквивалентна функции, возвращаемой вызовом `(stepper neighbours #{3} #{2 3})`. Эта функция высшего порядка `stepper` может использовать разные правила определения условий продолжения/зарождения/прекращения жизни и разные топологии, упоминавшиеся выше (гексагональная, трехмерная, ко-

нечная, сферическая, тороидальная, в виде ленты Мебиуса, и так далее). Например, клеточный автомат H.B2/S34 (гексагональная решетка, зарождение жизни при наличии двух живых соседей, продолжение жизни при наличии трех или четырех живых соседей) реализовать достаточно просто, как показано ниже:

```
(defn hex-neighbours
  [[x y]]
  (for [dx [-1 0 1] dy (if (zero? dx) [-2 2] [-1 1])]
    [(+ dx x) (+ dy y)]))

(def hex-step (stepper hex-neighbours #{2} #{3 4}))

;= ; эта конфигурация определяет осциллятор с периодом в 4 шага
(hex-step #{[0 0] [1 1] [1 3] [0 4]})
;= #{[1 -1] [2 2] [1 5]}
(hex-step *1)
;= #{[1 1] [2 4] [1 3] [2 0]}
(hex-step *1)
;= #{[1 -1] [0 2] [1 5]}
(hex-step *1)
;= #{[0 0] [1 1] [1 3] [0 4]}
```

Итак, четырехстрочная функция `stepper` является универсальной фабрикой для произвольных клеточных автоматов. Это оказалось возможным не благодаря отказу от индексов (потому что наша реализация на основе последовательности оказалась не универсальной), а за счет глубокого изменения структур данных, и перехода к использованию множеств, естественных идентификаторов и ассоциативных массивов (для отображения частот). Именно из-за того, что данное решение опирается на множества и естественные идентификаторы, его можно назвать «реляционным».

Примечание. Помимо обобщенного использования структур данных вместо конкретных индексов, между функциями `step` и `indexed-step` существуют и другие отличия: последняя действует в пределах конечной прямоугольной решетки, тогда как первая работает с бесконечной плоской решеткой. Однако мы легко можем реализовать создание `index-step` с помощью `stepper`, предположив, что `w` и `h` являются глобальными или локальными привязками ширины и высоты желаемой конечной решетки:

```
(stepper #(filter (fn [[i j]] (and (< -1 i w) (< -1 j h)))
                  (neighbours %)) #{2 3} #{3})
```

Генерация лабиринтов

Рассмотрим еще один пример: алгоритм генерации лабиринтов Уилсона (Wilson's maze generation algorithm)[1].

Алгоритм Уилсона – это алгоритм «вырезания». Он берет исходный «лабиринт», состоящий из клеток, окруженных с четырех сторон стенами, и вырезает в нем проходы, удаляя некоторые стены. Этот алгоритм имеет следующий принцип действия:

1. Выбирается случайная клетка и помечается как «посещенная».
2. Выбирается случайная клетка, еще не помеченная как «посещенная» – если такой клетки нет, возвращается готовый лабиринт.
3. Из вновь выбранной клетки случайным образом прокладывается маршрут, пока не будет встречена клетка, помеченная как «посещенная» – если во время прокладки маршрута одна и та же клетка встречается на пути несколько раз, для нее всегда запоминается направление, в котором эта клетка покидалась в последний раз.
4. Все клетки, встретившиеся во время прокладки маршрута, помечаются как «посещенные» и удаляются стены, соответствующие «направлениям выхода» из каждой клетки в последний раз.
5. Возврат к шагу 2.

В общем случае для представления лабиринта алгоритм генерации лабиринта использует матрицу, каждый элемент которой представляет собой битовую маску, указывающую, какие стены вокруг данной клетки остались на месте. Проницательный читатель может заметить, что при таком подходе состояние каждой стены хранится в двух местах – в смежных клетках, для которых эта стена является общей.

Кроме того, алгоритм Уилсона требует запоминать направление выхода для каждой клетки – еще один источник сложности, если попытаться впихнуть всю информацию в битовую маску, или «состояние клетки». Существует еще ряд причин, почему алгоритм Уилсона считается сложным для реализации. Однако, вооруженные языком Clojure и знанием особенностей «реляционного» моделирования, мы можем минимизировать сложности с помощью множеств, ассоциативных массивов и естественных идентификаторов!

---

[1] Кому интересна тема генераторов лабиринтов, я рекомендую обратиться к серии иллюстрированных статей Джеймиса Бака (Jamis Buck) по адресу: http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap.

Если внимательно изучить описание этого алгоритма, можно выделить несколько основных его элементов: клетки, признак, что клетка «посещалась», сам лабиринт, случайный маршрут и направление выхода. Посмотрим, как лучше представить их на языке Clojure.

На данный момент для вас не будет сюрпризом, что клетки должны быть представлены вектором с координатами, `[x y]`.

Стоп! Если представление клетки сводится к ее координатам, как тогда хранить дополнительный признак, что клетка «посещалась»? Ответ прост: этот признак будет храниться отдельно от координат, потому что он не принадлежит к ним, координаты – это лишь идентификатор (естественный). Следовательно, необходимо создать множество координат посещавшихся клеток.

Сам лабиринт состоит из клеток, окруженных стенами, и каждая стена разделяет две смежные клетки, поэтому стены должны быть представлены парами координат смежных ячеек, то есть, вектор `[[0 0] [1 0]]` в этом случае мог бы обозначать стену между клетками с координатами `[0 0]` и `[1 0]`. Беда в том, что ту же стену можно представить вектором `[[1 0] [0 0]]`. Поскольку порядок следования ячеек в парах не имеет значения, эти пары следует хранить в виде неупорядоченной коллекции... такой как множество. Таким образом множество `#{[0 0] [1 0]}` будет являться уникальным естественным идентификатором единственной стены. Лабиринт – это множество стен, поэтому вполне естественно будет представить его в виде множества[1]!

Маршрут представить совсем несложно – это обычная последовательность координат клеток. Однако в данном случае понятие *последовательность* имеет более широкое толкование: для этой цели подойдет любой последовательный тип, включая сами последовательности, а также векторы и списки.

И последний элемент – направление выхода. Направление выхода определяется клеткой, откуда осуществляется выход, и клеткой, куда производится вход, то есть, направление – это пара координат клеток `[from to]`, но, в отличие от стен, порядок следования координат имеет значение, поэтому для преставления направления выхода будут использоваться векторы.

---

[1] Лабиринт в этом случае будет представлять собой множество двухэлементных множеств с координатами клеток. Не позволяйте структуре данных вызывать головокружение: не заглядывайте на всю глубину вложенности, в каждый конкретный момент думайте только об одном уровне.

Теперь, определившись со структурами данных, можно перейти к программному коду:

```
(defn maze
  "Возвращает случайный вырезанный лабиринт; стены - это множество
   2-элементных множеств #{a b}, где a и b - координаты.
   Возвращаемый лабиринт - это множество удаленных стен."
  [walls]
  (let [paths (reduce (fn [index [a b]]                                    ❶
                        (merge-with into index {a [b] b [a]}))
                {} (map seq walls))                                        ❷
        start-loc (rand-nth (keys paths))]                                 ❸
   (loop [walls walls
          unvisited (disj (set (keys paths)) start-loc)]                   ❹
     (if-let [loc (when-let [s (seq unvisited)] (rand-nth s))]             ❺
       (let [walk (iterate (comp rand-nth paths) loc)                      ❻
             steps (zipmap (take-while unvisited walk) (next walk))]       ❼❽
         (recur (reduce disj walls (map set steps))                        ❾
           (reduce disj unvisited (keys steps))))
       walls))))
```

❶ `paths` – это индекс (ассоциативный массив) с направлениями переходов из одной клетки в другую (в виде векторов, см. ❻).

❷ `(map seq walls)` преобразует множество стен в последовательность, чтобы их можно было деструктурировать с помощью `[a b]`[1].

❸ Конструктивно `(keys path)` содержит все координаты, таким образом `rand-nth` возвращает координаты начальной клетки для построения маршрута.

❹ Вместо *множества посещавшихся клеток*, в программе используется его дополнение – *множество не посещавшихся клеток*, потому что в противном случае программный код получился бы сложнее (см. ❺ и ❼).

❺ Здесь вызов `seq` преследует две цели: убедиться, что множество непустое, и получить последовательное представление, чтобы можно было задействовать `rand-nth`. Если бы вместо *не посещавшихся* использовалось множество *посещавшихся* клеток, вызов `(seq unvisited)` пришлось бы заменить на `(seq (remove visited (keys paths)))`.

❻ Вызов `(iterate (comp rand-nth paths) loc)` выполняет бесконечный обход клеток в случайных направлениях: в вызов передаются координаты клетки, к ним применяется `paths`, чтобы получить вектор смеж-

[1] Деструктуризацию можно было бы выполнить непосредственно, с помощью `[& [a b]]`, но этот прием выглядит грубовато.

ных клеток, из которых с помощью `rand-nth` выбирается одна. Если бы `paths` возвращала множества, а не последовательность (например, вектор), тогда пришлось бы использовать форму `(comp rand-nth seq paths)`.

❼ `(take-while unvisited walk)` обеспечивает обход, пока не будет встречена посещавшаяся клетка (но не включает ее). Если бы в реализации использовалось множество посещавшихся клеток, тогда вместо `(take-while unvisited walk)` пришлось бы использовать `(take-while (complement visited) walk)`.

❽ `(next walk)` – выполняется до бесконечности, но `(take-while unvisited walk)` нет, поэтому `zipmap` получает только первые *n* элементов `(next walk)` (где *n* – это `(count (take-while unvisited walk))`). Первые *n* элементов, возвращаемых `(next walk)`, – это последовательность координат точек случайного маршрута без начальной точки, но с первой попавшейся на пути посещавшейся клеткой. Поскольку две последовательности оказываются сдвинуты относительно друг друга на один элемент, каждая получившаяся пара ключ/значение превращается в определение *направления*. При создании ассоциативного массива из этих пар, для каждого данного ключа будет сохраняться только самое последнее *направление выхода*. *Элементы в окончательном ассоциативном массиве хранят последние направления выхода* для каждой клетки, встретившейся во время обхода.

❾ `(map set steps)` преобразует направления (элементы ассоциативного массива) в стены в лабиринте (множества), которые нужно удалить.

Для проверки этой замечательной реализации необходимо создать две вспомогательные функции: `grid`, создающую заготовку лабиринта, где каждая клетка окружена четырьмя стенами, и `draw`, отображающая лабиринт (в данном случае с помощью Swing `JFrame`), как показано на рис. 3.10:

```
(defn grid
  [w h]
  (set (concat
         (for [i (range (dec w)) j (range h)] #{[i j] [(inc i) j]})
         (for [i (range w) j (range (dec h))] #{[i j] [i (inc j)]}))))

(defn draw
  [w h maze]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
      (doto (proxy [javax.swing.JPanel] []
```

```
            (paintComponent [^java.awt.Graphics g]
              (let [g (doto ^java.awt.Graphics2D (.create g)
                        (.scale 10 10)
                        (.translate 1.5 1.5)
                        (.setStroke (java.awt.BasicStroke. 0.4)))]
                (.drawRect g -1 -1 w h)
                (doseq [[[xa ya] [xb yb]] (map sort maze)]
                  (let [[xc yc] (if (= xa xb)
                                  [(dec xa) ya]
                                  [xa (dec ya)])]
                    (.drawLine g xa ya xc yc))))))
      (.setPreferredSize (java.awt.Dimension.
                          (* 10 (inc w)) (* 10 (inc h))))))
  .pack
  (.setVisible true)))

(draw 40 40 (maze (grid 40 40)))
```

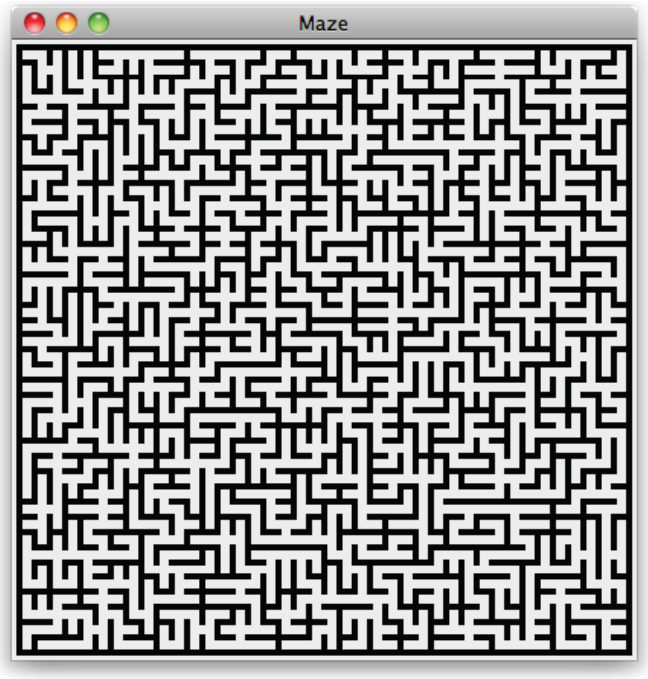

Рис. 3.10. Вид сгенерированного лабиринта

Истинный алгоритм Уилсона

В действительности мы немного схитрили и функция `maze` не является точной реализацией алгоритма Уилсона.

Когда при прокладке случайного маршрута встречается клетка, имеющаяся на графе (в лабиринте), мы добавляем целое дерево, состоящее из всех клеток, встретившихся на пути, а не ветвь дерева, тянущуюся от начальной точки до конечной. Очевидно, что наш алгоритм быстрее, потому что за один раз он добавляет в лабиринт больше клеток. Однако выигрышной особенностью алгоритма Уилсона является то обстоятельство, что все лабиринты являются одинаково вероятностными.

Эмпирическим путем (сравнением распределения лабиринтов, сгенерированных обоими алгоритмами, нашим и Уилсона) установлено, что эта особенность присуща и нашему варианту, но мы не доказали это формально и не определяли его сложность относительно времени. Это мы оставляем читателям, как самостоятельное упражнение[1].

За нашей хитростью стоит интересная история: мы почти случайно выбрали этот алгоритм – его проще было реализовать на языке Clojure, и он так и просился, чтобы реализовать его.

Реализация истинного алгоритма Уилсона мало чем отличается. Она содержит всего две дополнительные строки:

```
(defn wmaze
  «Оригинальный алгоритм Уилсона.¨
  [walls]
  (let [paths (reduce (fn [index [a b]]
                        (merge-with into index {a [b] b [a]}))
                {} (map seq walls))
        start-loc (rand-nth (keys paths))]
    (loop [walls walls unvisited (disj (set (keys paths)) start-loc)]
      (if-let [loc (when-let [s (seq unvisited)] (rand-nth s))]
        (let [walk (iterate (comp rand-nth paths) loc)
              steps (zipmap (take-while unvisited walk) (next walk))
              walk (take-while identity (iterate steps loc))      ❶
              steps (zipmap walk (next walk))]                    ❷
          (recur (reduce disj walls (map set steps))
            (reduce disj unvisited (keys steps))))
        walls))))
```

❶ Трассирует только одну «ветвь» случайного маршрута, начиная от клетки `loc`.

❷ Преобразует маршрут в ассоциативный массив элементов `[from-loc to-loc]`.

[1] Нам тоже интересно будет узнать ответ!

Говоря формальным языком, алгоритм Уилсона генерирует связующие деревья графов. В оригинальной статье[1], приводится псевдокод реализации, в котором, несмотря на его императивность, используются множества и ассоциативные массивы, но он опирается на синтетические идентификаторы (номера узлов). Эта реализация по-прежнему расценивается как весьма выразительная и ясная, но была забыта большинством разработчиков генераторов лабиринтов.

В качестве награды, возможно вы заметили, что, что подобно функции `step` выше, `maze` не зависит от фактического значения, определяющего клетку[2]. Как следствие, `maze` может генерировать лабиринты с любой топологией: гексагональные, N-мерные и так далее. В качестве доказательства такой универсальности, ниже представлены функции создания пустых и отображения готовых лабиринтов с гексагональной сеткой, изображенного на рис. 3.11:

```
(defn hex-grid
  [w h]
  (let [vertices (set (for [y (range h) x (range (if (odd? y) 1 0) (* 2 w) 2)]
                        [x y]))
        deltas [[2 0] [1 1] [-1 1]]]
    (set (for [v vertices d deltas f [+ -]
               :let [w (vertices (map f v d))]
               :when w] #{v w}))

(defn- hex-outer-walls
  [w h]
  (let [vertices (set (for [y (range h) x (range (if (odd? y) 1 0) (* 2 w) 2)]
                        [x y]))
        deltas [[2 0] [1 1] [-1 1]]]
    (set (for [v vertices d deltas f [+ -]
               :let [w (map f v d)]
               :when (not (vertices w))] #{v (vec w)})))
(defn hex-draw
  [w h maze]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
```

[1] Дэвид Брюс Уилсон (David Bruce Wilson), «Generating Random Spanning Trees More Quickly than the Cover Time» http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.8598.

[2] Если оно не равно `nil` или `false`.

```
    (doto (proxy [javax.swing.JPanel] []
            (paintComponent [^java.awt.Graphics g]
              (let [maze (into maze (hex-outer-walls w h))
                    g (doto ^java.awt.Graphics2D (.create g)
                        (.scale 10 10)
                        (.translate 1.5 1.5)
                        (.setStroke (java.awt.BasicStroke. 0.4
                                     java.awt.BasicStroke/CAP_ROUND
                                     java.awt.BasicStroke/JOIN_MITER)))
                    draw-line (fn [[[xa ya] [xb yb]]]
                                (.draw g
                                  (java.awt.geom.Line2D$Double.
                                   xa (* 2 ya) xb (* 2 yb))))]
                (doseq [[[xa ya] [xb yb]] (map sort maze)]
                  (draw-line
                    (cond
                      (= ya yb) [[(inc xa) (+ ya 0.4)]
                                 [(inc xa) (- ya 0.4)]]
                      (< ya yb) [[(inc xa) (+ ya 0.4)] [xa (+ ya 0.6)]]
                      :else [[(inc xa) (- ya 0.4)]
                             [xa (- ya 0.6)]]))))))
      (.setPreferredSize (java.awt.Dimension.
                          (* 20 (inc w)) (* 20 (+ 0.5 h))))))
  .pack
  (.setVisible true)))

(hex-draw 40 40 (maze (hex-grid 40 40)))
```

Реализовать функцию `maze` на языке Java, Python или Ruby возможно, но она получится хрупкой, поскольку будет опираться на изменяемые объекты[1]. В таком небольшом примере это обстоятельство, возможно, не будет представлять большую проблему, но в крупном проекте будет довольно сложно следовать этой дисциплине. Как вариант, можно предусмотреть защитные меры в виде глубокого копирования исходных данных, однако это может привести к снижению эффективности в рамках модуля. На этих языках *тяжело* реализуются подобные решения, тогда как на языке Clojure *тяжело* реализуются императивные решения, поскольку его основные возможности позволяют создавать реализации, подобные этим, в отличие от других языков.

---

[1] Этот недостаток можно было бы смягчить с помощью метода `freeze` в Ruby или использовать немодифицируемые обертки в Java.

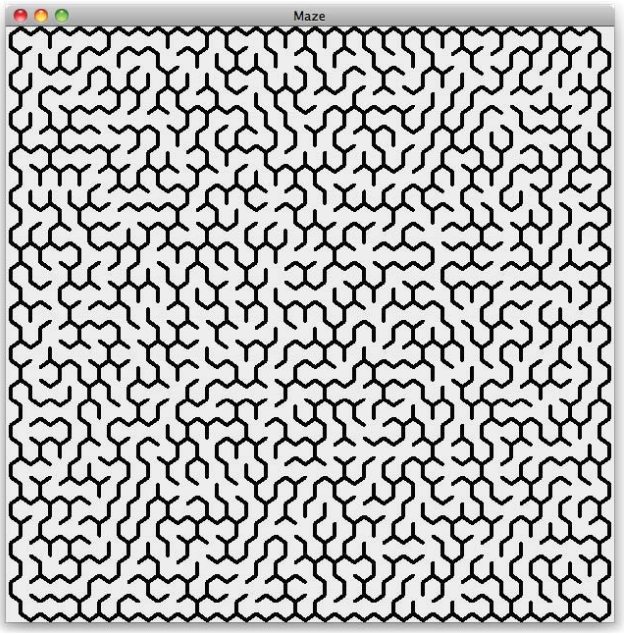

Рис. 3.11. Вид сгенерированного лабиринта

В общем случае, когда вы обнаруживаете, что создание программного кода продвигается тяжело, а сам он получается неуклюжим, чаще всего это означает, что вы воюете с языком, идя наперекор его естеству. В таких ситуациях высока вероятность, что вы сможете отыскать более красивое решение, переосмыслив свои структуры данных. Как говорил Брукс, код определяется моделью данных, поэтому красивый программный код на Clojure возможен только при тщательном обдумывании представления данных, что часто подразумевает применение естественных составных идентификаторов, множеств и ассоциативных массивов.

Навигация, изменение и зипперы (zippers)

Так как неизменяемость устраняет обратные ссылки, исключается возможность использовать их для навигации по деревьям. Типичным функциональным решением этой проблемы являются *зипперы* (zippers), реализацию которых можно найти в в пространстве имен `clojure.zip`.

Зиппер мало чем отличается от путеводной нити Ариадны, которая помогла Тесею найти выход из Лабиринта после битвы с Минотавром[1]: по сути это стек, где сохраняются все посещаемые узлы и их потомки. Это своеобразный курсор, являющийся одновременно механизмом навигации и редактирования. Вы можете перемещать зиппер, определять текущий узел и позицию, изменять дерево в текущей позиции и затем возвращаться обратно по пройденному маршруту, чтобы получить измененное дерево. Кроме того, подобно коллекциям в Clojure, зипперы являются сохраняемыми и неизменяемыми, поэтому операции перемещения или изменения зиппера возвращают новый зиппер, и не изменяют исходный зиппер или дерево.

Манипулирование зипперами

В пространстве имен `clojure.zip` имеется универсальная фабричная функция `zipper` и три более специализированные функции: `seq-zip` – для обработки вложенных последовательностей, `vector-zip` – для обработки вложенных векторов, и `xml-zip` – для обработки XML-данных, представленных с помощью `clojure.xml`. Ниже будет показано, как создавать собственные зипперы, а пока рассмотрим примеры использования зипперов, созданных с помощью `vector-zip`.

В число базовых функций, осуществляющих перемещение зиппера, входят: `up` (перемещение в сторону корня дерева), `down` (перемещение в сторону листьев), а также `left` и `right` для перемещения между соседними (братскими) узлами. Имеются также функции `prev` и `next`, выполняющие обход по принципу «сначала в глубину» (depth-first), и `leftmost` и `rightmost`, выполняющие переход к первому или последнему соседнему (братскому) узлу, но они не будут вызывать перемещение, если зиппер уже находится в соответствующем узле.

Для определения текущей позиции/узла, в пространстве имен `clojure.zip` имеются функции `node`, `branch?`, `children`, `lefts`, `rights` и `root`, которые, соответственно, возвращают: текущий узел, признак – является ли текущий узел ветвью[1], дочерние узлы (для ветвей) и все соседние (братские) узлы слева и справа от текущего узла. Функция `root` приобретает особую важность при изменении дерева с помощью

---

[1] http://ru.wikipedia.org/wiki/Ариадна.

[2] Ветвь – это узел, который *может* иметь дочерние узлы: ветвь может не иметь дочерних узлов.

зипперов, поскольку это единственный способ получить измененное дерево, отражающее все изменения, произведенные зиппером с момента его создания.

```
(require '[clojure.zip :as z])

(def v [[1 2 [3 4]] [5 6]])
;= #'user/v
(-> v z/vector-zip z/node)
;= [[1 2 [3 4]] [5 6]]
(-> v z/vector-zip z/down z/node)
;= [1 2 [3 4]]
(-> v z/vector-zip z/down z/right z/node)
;= [5 6]
```

С помощью `remove` можно удалить текущий узел, с помощью `replace` – заменить его другим узлом, с помощью `insert` – вставить дочерний узел перед текущим, а с помощью `append` вставить дочерний узел после текущего. Кроме того, с помощью `edit` можно вносить изменения в узел. Функция `edit` использует единообразную модель внесения изменений: помимо зиппера она принимает функцию *f* и дополнительные аргументы, а затем замещает текущий узел результатом применения к нему функции *f* с дополнительными аргументами.

Имеется также возможность создавать новые узлы на основе текущей позиции зиппера с помощью `makenode`. Однако эти узлы не будут включаться в итоговое дерево – для этого нужно использовать одну из функций, перечисленных выше.

```
(-> v z/vector-zip z/down z/right (z/replace 56) z/node)
;= 56
(-> v z/vector-zip z/down z/right (z/replace 56) z/root)          ❶
;= [[1 2 [3 4]] 56]
(-> v z/vector-zip z/down z/right z/remove z/node)                ❷
;= 4
(-> v z/vector-zip z/down z/right z/remove z/root)
;= [[1 2 [3 4]]]
(-> v z/vector-zip z/down z/down z/right (z/edit * 42) z/root)    ❸
;= [[1 84 [3 4]] [5 6]]
```

❶ `z/vector-zip` и `z/root` в паре действуют как границы «транзакции».
❷ `z/remove` перемещает зиппер в предыдущее местоположение при обходе «в глубину».
❸ Узел 2 умножается на 42.

Это практически все[1], что требуется знать о прикладном интерфейсе зипперов. Теперь можно посмотреть, как создавать собственные зипперы и как их можно использовать в паре с нашими лабиринтами.

Собственные зипперы

В общем случае зипперы создаются с помощью функции `zipper`, принимающей три другие функции вслед за корневым узлом структуры, к которой будет применяться зиппер:

- предикат, возвращающий `true`, если узел *может* иметь дочерние узлы;
- функция, возвращающая последовательность дочерних узлов для данного узла ветви;
- функция, возвращающая новый узел ветви для данного существующего узла, и последовательность дочерних узлов.

Попробуем реализовать зиппер для нашей модели данных: векторов, представляющих HTML-элементов. Первый элемент – это имя тега, второй может содержать ассоциативный массив атрибутов, и остальные – дочерние узлы, включая текстовые в виде строк. Таким образом, оба вектора, `[:h1 "zipper"]` и `[:a {:href "http://clojure.org/"} "Clojure"]`, являются допустимыми узлами. Эта модель достаточно нерегулярна, чтобы сделать пример реализации зиппера интереснее:

```
(defn html-zip [root]
  (z/zipper
    vector?
    (fn [[tagname & xs]]
      (if (map? (first xs)) (next xs) xs))
    (fn [[tagname & xs] children]
      (into (if (map? (first xs)) [tagname (first xs)] [tagname])
        children))
    root))
```

Поверх универсальных функций зипперов можно создавать вспомогательные, предметно-ориентированные функции, такие как `wrap`:

```
(defn wrap
  "Заключает текущий узел в указанный тег с атрибутами."
```

[1] Здесь не упомянута функция `end?`, проверяющая – достигнут ли конец при обходе «в глубину».

```
  ([loc tag]
    (z/edit loc #(vector tag %)))
  ([loc tag attrs]
    (z/edit loc #(vector tag attrs %))))

(def h [:body [:h1 "Clojure"]
              [:p "What a wonderful language!"]])
;= #'user/h
(-> h html-zip z/down z/right z/down (wrap :b) z/root)
;= [:body [:h1 "Clojure"] [:p [:b "What a wonderful language!"]]]
```

Создание собственных зипперов для вложенных структур данных не отличается особой сложностью. Однако, также можно создавать зипперы «только для чтения»[1] для иерархических данных, хранящихся не в виде иерархической структуры, такой как лабиринты, создававшиеся в разделе «Генерация лабиринтов», выше. В таких ситуациях особенно важным становится то обстоятельство, что аргументы функции `zipper` образуют замыкания, сохраняющие структуру данных, а узлы превращаются в естественные идентификаторы для этой структуры данных.

Зиппер Ариадны

Коль скоро мы говорили о лабиринтах и мифологических приключениях, можно попробовать использовать зипперы буквально как нить Ариадны и помочь Тесею выйти из лабиринта.

Для начала нужно создать лабиринт:

```
(def labyrinth (maze (grid 10 10)))
```

Но этот лабиринт состоит из одних стен, тогда как нас интересуют только проходы в нем:

```
(def labyrinth (let [g (grid 10 10)] (reduce disj g (maze g))))
```

Добавим персонажи:

```
(def theseus (rand-nth (distinct (apply concat labyrinth))))
(def minotaur (rand-nth (distinct (apply concat labyrinth))))
```

---

[1] Реализовать поддержку режима «для чтения и записи» можно, но это сложнее.

Эффективнее было бы определить их как `[(rand-int 10) (rand-int 10)]`, но тогда между определением лабиринта `labyrinth` и координатами персонажей образовалась бы тесная зависимость.

На первый взгляд координат вполне достаточно, чтобы идентифицировать местоположение Тесея, когда он будет искать Минотавра. Однако в действительности это не так: необходимо также запоминать координаты, куда мы направляемся, чтобы можно было отличать проходы, ведущие к новым, еще не исследованным комнатам, и откуда. Нам необходимо хранить направление движение, которое лучше представить в виде координат пары точек.

Итак, зиппер Ариадны будет интерпретировать направления как узлы:

```
(defn ariadne-zip
  [labyrinth loc]
  (let [paths (reduce (fn [index [a b]]
                        (merge-with into index {a [b] b [a]}))
                {} (map seq labyrinth))
        children (fn [[from to]]
                   (seq (for [loc (paths to)                    ❶
                              :when (not= loc from)]
                          [to loc])))]
    (z/zipper (constantly true)                                 ❷
      children
      nil                                                       ❸
      [nil loc])))                                              ❹
```

❶ В документации с описанием функции `zipper` говорится, что ей в виде аргумента должна быть передана функция `children`, возвращающая *последовательность*, а не произвольный последовательный тип, поэтому к возвращаемому значению необходимо применить `seq`.

❷ В качестве предиката `branch?` используется `(constantly true)`, потому что из любой комнаты в лабиринте можно куда-то выйти.

❸ В качестве фабричной функции для создания новых узлов передается `nil`, потому что зиппер будет использоваться исключительно для навигации: он не позволяет вносить изменения.

❹ `[nil loc]` – это начальное направление, корень дерева, описывающего путь Тесея.

Теперь осталось лишь выполнить поиск в глубину по лабиринту, чтобы отыскать путь к Минотавру:

```
(->> theseus
     (ariadne-zip labyrinth)
     (iterate z/next)
     (filter #(= minotaur (second (z/node %))))
     first z/path
     (map second))
([3 9] [4 9] [4 8] [4 7] [4 6] [5 7] [5 6] [5 5] [5 4]
 [5 8] [6 8] [6 7] [6 6] [6 5] [7 6] [8 6] [9 6] [9 5]
 [9 4] [9 3] [9 2] [9 1] [9 0] [8 2] [8 1] [8 0] [7 0]
 [6 0] [7 1] [7 2] [6 2] [6 1] [5 1] [4 1] [4 0] [5 0]
 [3 0] [4 2] [5 2] [3 2] [3 3] [4 3] [4 4] [4 5] [3 5])
```

Когда Минотавр будет побежден, Тесей сможет отыскать дорогу обратно – к начальной точке!

Путь, проделанный Тесеем, можно показать на экране. Для этого сначала изменим функцию `draw`, добавив поддержку дополнительного аргумента: фактического пути от Тесея к Минотавру.

```
(defn draw
  [w h maze path]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
      (doto (proxy [javax.swing.JPanel] []
              (paintComponent [^java.awt.Graphics g]
                (let [g (doto ^java.awt.Graphics2D (.create g)
                          (.scale 10 10)
                          (.translate 1.5 1.5)
                          (.setStroke (java.awt.BasicStroke. 0.4)))]
                  (.drawRect g -1 -1 w h)
                  (doseq [[[xa ya] [xb yb]] (map sort maze)]
                    (let [[xc yc] (if (= xa xb)
                                    [(dec xa) ya]
                                    [xa (dec ya)])]
                      (.drawLine g xa ya xc yc)))
                  (.translate g -0.5 -0.5)
                  (.setColor g java.awt.Color/RED)
                  (doseq [[[xa ya] [xb yb]] path]          ❶
                    (.drawLine g xa ya xb yb)))))
        (.setPreferredSize (java.awt.Dimension.
                             (* 10 (inc w)) (* 10 (inc h))))))
  .pack
  (.setVisible true)))
```

❶ `path` – это коллекция пар местоположений, поэтому она отличается от ранее вычисленного пути, хранящего только координаты точек, возвращаемых (`map second`).

Теперь весь миф целиком можно рассказать на языке Clojure:

```
(let [w 40, h 40
      grid (grid w h)
      walls (maze grid)
      labyrinth (reduce disj grid walls)
      places (distinct (apply concat labyrinth))
      theseus (rand-nth places)
      minotaur (rand-nth places)
      path (->> theseus
             (ariadne-zip labyrinth)
             (iterate z/next)
             (filter #(= minotaur (first (z/node %))))
             first z/path rest)]    ❶
  (draw w h walls path))
```

❶ Вместо `(map second)` здесь используется `rest`, потому что первая пара координат не соответствует инициализации `ariadne-zip` значением `[nil theseus]`.

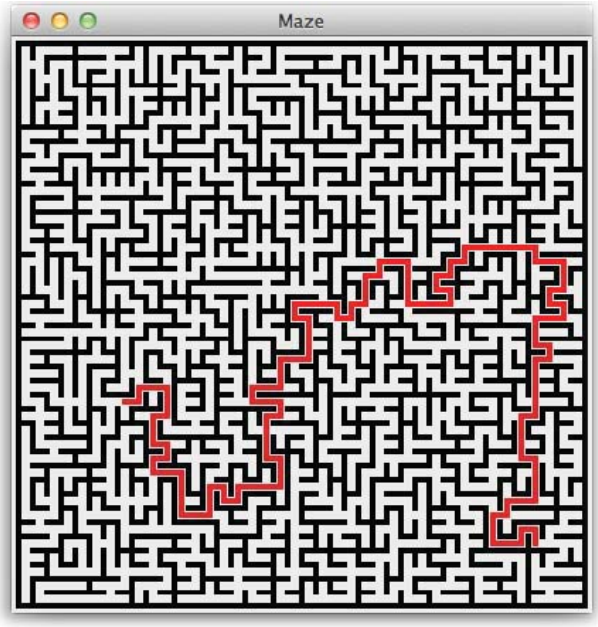

Рис. 3.12. Путь Тесея к Минотавру

В заключение

Абстракции коллекций в Clojure и реализации структур данных находятся в самом сердце языка и определяют его возможности, характер и мировоззрение гораздо больше, чем все остальное. Понимание их семантики, механики и идиоматических приемов использования позволяет выжать все возможное из функционального стиля программирования на языке Clojure, и дает возможность лучше понять все остальные части языка Clojure, зависящие от них.

Глава 4. Конкуренция и параллелизм

Создание многопоточных программ – одна из сложнейших задач, с которыми сталкиваются программисты. Такие программы трудно поддаются анализу и часто проявляют недетерминированное поведение: типичная программа, использующая механизмы конкурентного использования данных, иногда может давать разные результаты для одних и тех же исходных данных из-за нарушения порядка выполнения, которое может также приводить к состоянию гонки за ресурсами (race conditions) и взаимоблокировкам (deadlocks). Некоторые из этих состояний сложны в обнаружении и все они тяжелы в отладке.

Большинство языков программирования предоставляет ничтожно мало простых и понятных механизмов управления конкуренцией. Потоки выполнения и блокировки, во всех своих проявлениях, часто являются единственным доступным инструментом, и мы нередко вынуждены бороться с разнообразными сложностями, чтобы обеспечить корректное и эффективное их использование. В каком порядке должны приобретаться и освобождаться блокировки? Должен ли читающий поток приобретать блокировку для чтения значения, которое может изменяться другим потоком выполнения? Как обеспечить всестороннее тестирование многопоточной программы, опирающейся на использование блокировок? Спираль сложности раскручивается очень быстро и вот вы уже вынуждены заниматься отладкой гонки за ресурсами, возникающей только в эксплуатационном окружении, или взаимоблокировкой, проявляющейся на *этой* машине и не проявляющейся на *той*.

Подход на основе использования низкоуровневых потоков выполнения, блокировок и их производных, ничем особенным не выделяющихся, как единственное решение проблем конкуренции, резко контрастирует с постоянным стремлением программистов к более

эффективным абстракциям, снижающим вероятность появления ошибок. Ответ Clojure на эти проблемы имеет множество граней:

1. Как уже говорилось в главе 2, уменьшить объем изменяемого состояния в программах можно с помощью неизменяемых значений и коллекций, обладающих надежной семантикой и высокой эффективностью.
2. При необходимости поддержки состояния, изменяющегося во времени, в сочетании с конкурирующими потоками выполнения, есть возможность изолировать это состояние и ограничить способы его изменения. Это – основная область применения *ссылочных типов* языка Clojure, о которых будет рассказываться ниже.
3. Когда не остается выбора – и вы готовы пренебречь преимуществами ссылочных типов – легко можно вернуться к использованию обычных блокировок, потоков выполнения и высококачественного API, имеющегося в языке Java.
4. В Clojure нет универсального рецепта, делающего конкурентное программирование тривиальным, но в нем действительно имеются некоторые новейшие и ныне уже проверенные на практике инструменты, упрощающие реализацию параллельных вычислений и повышающие их надежность.

Сдвиг вычислений в пространстве и времени

В Clojure имеется ряд механизмов – `delay`, `future` и `promise` – позволяющих управлять фактическим моментом выполнения вычислений. Строго говоря, только механизм `future` имеет отношение к параллельному выполнению кода, однако все три часто используются при реализации той или иной семантики и механики параллельного выполнения.

delay

Конструкция `delay` откладывает выполнение некоторого блока кода до момента его разыменования:

```
(def d (delay (println "Running...")
              :done!))
;= #'user/d
```

```
(deref d)
; Running...
;= :done!
```

Примечание. Абстракция `deref` определена в виде интерфейса `clojure.lang.IDeref`. Любой тип, реализующий его, выступает в роли контейнера для хранения значения. Извлечь вычисленное значение можно с помощью `deref` или синтаксиса `@`[1]. Операция разыменования в языке Clojure может применяться ко многим объектам, включая `delay`, `future`, `promise` и всем ссылочным типам (атомам, ссылкам, агентам и переменным). Все они будут рассматриваться в этой главе.

Того же эффекта можно добиться с помощью простой функции:

```
(def a-fn (fn []
            (println "Running...")
            :done!))
;= #'user/a-fn
(a-fn)
; Running...
;= :done!
```

Однако конструкция `delay` предоставляет пару весьма привлекательных особенностей.

Значение определения `delay` вычисляется *только один раз* и сохраняется для последующего использования. То есть, при последующих обращениях с помощью `deref` код повторно выполняться не будет[2] и всегда будет возвращаться значение, вычисленное ранее:

```
@d
;= :done!
```

Как следствие, разные потоки могут безопасно выполнять первую попытку разыменования значения `delay` – все они будут заблоки-

---

[1] Синтаксис `@foo` практически всегда предпочтительнее конструкции `(deref foo)`, за исключением случаев, когда `deref` применяется с функциями высшего порядка (например, чтобы выполнить все отложенные вычисления в последовательности) или при использовании возможности `deref` ограничивать предельное время ожидания, доступной только в механизмах `promise` и `future`.

[2] И, соответственно, не будет вызывать возможные побочные эффекты.

рованы, пока код не выполнится (только один раз!) и значение не станет доступно.

Когда в программе потребуется определить значение, которое может не потребоваться или его вычисление является дорогостоящей операцией, для оптимизации можно использовать конструкцию `delay`, откладывая вычисления до момента, когда конечный пользователь будет готов «заплатить» цену за вычисления.

Пример 4.1. Откладывание выполнения операций с помощью delay

```
(defn get-document
  [id]
  ; ... выполняются операции по извлечению метаданных документа ...
  {:url "http://www.mozilla.org/about/manifesto.en.html"
   :title "The Mozilla Manifesto"
   :mime "text/html"
   :content (delay (slurp
                    "http://www.mozilla.org/about/manifesto.en.html"))})    ❶
;= #'user/get-document
(def d (get-document "some-id"))
;= #'user/d
d
;= {:url "http://www.mozilla.org/about/manifesto.en.html",
;= :title "The Mozilla Manifesto",
;= :mime "text/html",
;= :content #<Delay@2efb541d: :pending>}                                    ❷
```

❶ `delay` можно использовать, чтобы отложить выполнение потенциально дорогостоящих операций или получение дополнительных данных.

❷ Этот отложенный программный код не будет выполнен, пока мы (или вызывающая программа) не попытается разыменовать данное значение.

Некоторым частям программы не нужно содержимое документа – им вполне достаточно метаданных, и с помощью `delay` можно избежать затрат на извлечение этого содержимого. Другие части приложения могут требовать получение содержимого в обязательном порядке, а третьи – использовать его, *только если оно уже доступно*. Последний случай можно реализовать с помощью предиката `realized?`, проверяющего доступность значения, объявленного с помощью `delay`:

```
(realized? (:content d))
;= false
@(:content d)
;= "<!DOCTYPE html><html>..."
(realized? (:content d))
;= true
```

Примечание. Обратите внимание, что предикат `realized?` можно также использовать вместе с формами `future`, `promise` и «ленивыми» последовательностями.

`realized?` позволяет немедленно использовать данные, возвращаемые значением `delay`, если оно уже было разыменовано, или отказаться от дорогостоящих вычислений, если данные в текущий момент не являются строго необходимыми.

Механизм future

Прежде чем переходить к исследованию более сложных тем, таких как ссылочные типы, программисты часто задаются вопросом: «Как создать новый поток выполнения и запустить в нем некоторый программный код?». Для этого можно использовать механизм многопоточного выполнения, встроенный в JVM (см. раздел «Механизмы параллельного выполнения в Java», ниже), но Clojure предлагает более удобный и надежный способ – механизм `future`.

Программный код в определении `future` выполняется в другом потоке[1]:

```
(def long-calculation (future (apply + (range 1e8))))
;= #'user/long-calculation
```

`future` возвращает управление немедленно, позволяя текущему потоку выполнения (такому как оболочка REPL) продолжить работу. Результат вычислений можно получить позже, выполнив операцию разыменования:

```
@long-calculation
;= 4999999950000000
```

[1] В вашем распоряжении имеется еще один инструмент – `future-call` – позволяющий запускать в другом потоке выполнения функции без аргументов.

Как и в случае с механизмом `delay`, разыменование значения, полученного вызовом `future`, заблокирует вызывающий поток, если вычисления еще не завершились, то есть, следующее выражение заблокирует оболочку REPL на пять секунд:

```
@(future (Thread/sleep 5000) :done!)
;= :done!
```

Как и при использовании механизма `delay`, `future` сохраняет вычисленное значение, которое буде возвращаться немедленно при следующих попытках разыменования с помощью `deref`.

В отличие от `delay`, в форме разыменования значения `future` можно определить предельное время ожидания и значение по умолчанию, которое будет возвращаться по истечении предельного времени ожидания[1]:

```
(deref (future (Thread/sleep 5000) :done!)
       1000
       :impatient!)
;= :impatient!
```

Механизм `future` часто применяется для упрощения работы с прикладными интерфейсами, использующими в своей работе различные аспекты, связанные с многопоточным выполнением. Например, допустим известно, что все пользователи функции `get-document` из примера 4.1 обязательно будут получать значение `:content`. Тогда первое, что приходит в голову, – реализовать синхронное получение содержимого документа в рамках вызова `get-document`, но в этом случае при каждом вызове функции пришлось бы ждать, пока содержимое не будет получено полностью, даже если оно не нужно немедленно. Вместо этого значение `:content` можно получить с помощью механизма `future` – он запустит извлечение содержимого в другом потоке выполнения, позволив вызывающему потоку продолжить работу, не приостанавливаясь до окончания операции ввода/вывода. Когда позднее программа попытается разыменовать значение `:content`, она заблокируется (если вообще заблокируется), но уже на меньшее время, потому что операция извлечения содержимого к этому моменту уже будет на пути к завершению.

---

[1] Эта возможность недоступна при использовании синтаксического сахара `@`.

```
(defn get-document
[id]
; ... выполняются операции по извлечению метаданных документа ...
{:url "http://www.mozilla.org/about/manifesto.en.html"
 :title "The Mozilla Manifesto"
 :mime "text/html"
 :content (future (slurp "http://www.mozilla.org/about/manifesto.en.html"))})  ❶
```

❶ Здесь мы заменили `delay` на `future`, это единственное отличие от примера 4.1.

Такое решение не требует вносить изменения на стороне клиентов (поскольку они лишь разыменовывают значение `:content`), но, если вызывающая программа всегда будет получать содержимое документа, это небольшое усовершенствование сможет существенно улучшить производительность.

В сравнении со встроенной поддержкой многопоточного выполнения, механизм `future` имеет несколько важных преимуществ:

1. Вычисления производятся в рамках пула потоков, используемого также потенциально блокирующими операциями агентов (см. раздел «Агенты» ниже). Такая группировка ресурсов может повысить эффективность механизма `future` в сравнении с традиционным механизмом управления потоками.
2. С применением механизма `future` программный код получается более кратким.
3. Значения, возвращаемые формой `future`, являются экземплярами `java.util.concurrent.Future`, что упрощает взаимодействие с Java API.

Механизм promise

Механизм `promise` использует значительную часть механики `delay` и `future`: значение, возвращаемое формой `promise`, может разыменовываться с дополнительным параметром, ограничивающим максимальное время ожидания, операция разыменования значения `promise` будет блокироваться, пока значение не будет получено и `promise` может иметь только одно значение. Однако, в отличие от `delay` и `future`, значения `promise` не связаны с каким-либо программным кодом:

```
(def p (promise))
;= #'user/p
```

promise определяет пустой контейнер, куда позднее можно записать значение с помощью deliver:

```
(realized? p)
;= false
(deliver p 42)
;= #<core$promise$reify__1707@3f0ba812: 42>
(realized? p)
;= true
@p
;= 42
```

Этой своей особенностью promise напоминает одноразовый канал с единственным значением: запись данных в канал выполняется с помощью deliver, а чтение – с помощью deref. Такие программные объекты иногда называют *потоковыми переменными* (dataflow variables) и являются основой приема *декларативной конкуренции* (declarative concurrency). Этот механизм определяет стратегию, где отношения между конкурирующими процессами явно определяются так, что результаты вычисляться по мере готовности исходных данных, что обеспечивает определенность поведения. Ниже приводится простой пример использования трех значений promise:

```
(def a (promise))
(def b (promise))
(def c (promise))
```

Определить взаимосвязи между этими объектами promise можно с помощью объекта future, использующего (еще не вычисленные) значения некоторых объектов promise для вычисления значения другого объекта:

```
(future
  (deliver c (+ @a @b))
  (println "Delivery complete!"))
```

В данном случае значение c не будет получено, пока не появятся оба значения, a и b (то есть, пока предикат realized? не вернет true). До того момента форма future, осуществляющая запись значения в c, будет заблокирована на операциях разыменования a и b. Обратите внимание, что попытка разыменовать c (без ограничения времени

ожидания) со значениями promise в текущем состоянии «навечно» заблокирует поток выполнения оболочки REPL.

В большинстве случаев, когда применяются потоковые переменные, в программе существуют другие потоки выполнения, производящие необходимые вычисления, которые в конечном итоге запишут значения в a и b. Мы можем сымитировать процесс записи значений из REPL[1]. Как только оба значения, a и b, будут записаны, выполнение операций разыменования в future окажутся разблокированы и появится возможность записать значение в c:

```
(deliver a 15)
;= #<core$promise$reify__5727@56278e83: 15>
(deliver b 16)
; Delivery complete!
;= #<core$promise$reify__5727@47ef7de4: 16>
@c
;= 31
```

Внимание. Механизм promise не способен выявлять циклические зависимости. Это означает, что выражение (deliver p @p), где p определяется, как promise, будет заблокировано «навечно».
Однако подобные ситуации не являются тупиковыми и имеют решение:

```
(def a (promise))
(def b (promise))
(future (deliver a @b))      ❶
(future (deliver b @a))
(realized? a)                ❷
;= false
(realized? b)
;= false
(deliver a 42)               ❸
;= #<core$promise$reify__5727@6156f1b0: 42>
@a
;= 42
@b
;= 42
```

❶ Механизм future используется здесь, чтобы не заблокировать REPL.
❷ Значения a и b еще не записаны.
❸ Запись значения в a разблокирует операцию разыменования. Очевидно, что операция (deliver a @b) потерпит неудачу (и вернет nil), но операция (deliver b @a) выполнится успешно.

[1] Фактически – *из другого* потока выполнения!

Непосредственным практическим применением механизма promise является упрощение создания синхронного API на основе обратных вызовов. Допустим, имеется функция, принимающая другую функцию для обратного вызова:

```
(defn call-service
  [arg1 arg2 callback-fn]
  ; ...выполняет обслуживание и в конечном итоге
  ;     вызывает callback-fn с результатами...
  (future (callback-fn (+ arg1 arg2) (- arg1 arg2))))
```

Чтобы использовать результаты работы этой функции в синхронном программном коде, необходимо передать ей функцию обратного вызова и затем использовать самые разные (порой неприятные) способы ожидания передачи результатов функции обратного вызова. Как вариант, поверх асинхронного API, основанного на обратных вызовах, можно написать простую обертку, использующую способность механизма promise блокироваться в операции deref, чтобы обеспечить принудительную синхронизацию. Допустим на мгновение, что все необходимые вам асинхронные функции принимают функции обратного вызова в последнем аргументе, тогда подобную обертку можно реализовать в виде обобщенной функции высшего порядка:

```
(defn sync-fn
  [async-fn]
  (fn [& args]
    (let [result (promise)]
      (apply async-fn (conj (vec args) #(deliver result %&)))
      @result)))

((sync-fn call-service) 8 7)
;= (15 1)
```

Параллельная обработка по невысокой цене

Чуть ниже мы займемся исследованием всех механизмов поддержки конкуренции в языке Clojure, один их которых – агенты (agents) – можно использовать для эффективного управления нагрузками. Однако иногда может потребоваться обеспечить параллельное выполнение операций с минимумом церемоний.

Параллелизм и конкуренция

Чтобы в процессе обсуждения конкуренции и параллелизма у вас не возникало ощущения, что это одно и то же, попробуем отделить эти два понятия друг от друга.

Конкуренция (concurrency) – это координация действий нескольких потоков выполнения, обычно действующих поочередно, которые обращаются к общим данным или изменяют их.

Параллелизм (parallelism) также имеет отношение к совместно используемым данным, но в первую очередь является приемом более эффективного использования доступных ресурсов (обычно вычислительных, но иногда и других, таких как полоса пропускания), позволяющим повысить производительность операций. Цель приемов распараллеливания обычно состоит в том, чтобы максимально увеличить окно монопольного доступа к состоянию (или к отдельным его частям) и уменьшить накладные расходы на координацию действий. Вместо поддержки чередования потоков выполнения, приемы распараллеливания применяются для разделения вычислений на операции, которые могут выполняться одновременно – иногда на разных ядрах центрального процессора (CPU), а иногда даже на разных компьютерах.

Высокая гибкость абстракции последовательностей[1] в Clojure значительно упрощает реализацию многих процедур, обрабатывающих последовательности. Например, представьте, что имеется функция, использующая регулярные выражения для поиска телефонных номеров в строках:

```
(defn phone-numbers
  [string]
  (re-seq #"(\d{3})[\.-]?(\d{3})[\.-]?(\d{4})" string))
;= #'user/phone-numbers
(phone-numbers " Sunil: 617.555.2937, Betty: 508.555.2218")
;= (["617.555.2937" "617" "555" "2937"] ["508.555.2218" "508" "555" "2218"])
```

Функция достаточно простая, быстрая, эффективная и легко может применяться к любым последовательностям строк. Последовательности могут загружаться с диска с помощью `slurp` и `file-seq`, поступать в виде сообщений из очереди или извлекаться в виде больших блоков текста из базы данных. Для простоты смоделируем последовательность из 100 строк, каждая из которых имеет размер 1 Мбайт и содержит в конце несколько телефонных номеров:

[1] Обсуждалась в разделе «Последовательности», в главе 3.

```
(def files (repeat 100
                   (apply str
                     (concat (repeat 1000000 \space)
                             "Sunil: 617.555.2937, Betty: 508.555.2218"))))
```

Посмотрим, насколько быстро можно извлечь все телефонные номера из этих «файлов»:

```
(time (dorun (map phone-numbers files)))    ❶
; "Elapsed time: 2460.848 msecs"
```

❶ Функция `dorun` используется здесь, чтобы полностью реализовать «ленивую» последовательность, возвращаемую функцией `map`, и одновременно отбросить результаты этой реализации, чтобы исключить вывод всех найденных телефонных номеров в окне REPL.

Однако эту операцию легко можно распараллелить. Функция `map` имеет родственную ей функцию `pmap`, распараллеливающую применение функции к значениям в последовательности и возвращающую «ленивую» последовательность результатов, как и сама функция `map`:

```
(time (dorun (pmap phone-numbers files)))    ❶
; "Elapsed time: 1277.973 msecs"
```

При выполнении на компьютере с двухъядерным процессором применение `pmap` вместо `map` уменьшило время выполнения операции почти вдвое, по сравнению с предыдущим примером. В данном конкретном случае и с этим набором данных на четырехъядерном процессоре вполне можно ожидать примерно четырехкратного уменьшения времени выполнения, и так далее. Совсем неплохо, если учесть, что мы добавили всего один символ в имени функции! Все это может походить на волшебство, однако это не так. Просто функция `pmap` использует механизм `future` – откалиброванный по количеству доступных ядер процессора – для распределения вычислений, производимых функцией `phone-numbers`, по ядрам процессора.

Такой подход дает положительные результаты во многих ситуациях, но это не значит, что `pmap` можно применять бездумно. Распараллеливание операций, как в данном примере, сопряжено с определенными накладными расходами. Если операции, которые предполагается распараллелить, выполняются очень быстро, накладные расходы могут превосходить затраты вычислительных

ресурсов на выполнение фактической работы. Ниже приводится пример, где использование функции `pmap` вместо `map` замедляет выполнение операции:

```
(def files (repeat 100000
                   (apply str
                     (concat (repeat 1000 \space)
                             "Sunil: 617.555.2937, Betty: 508.555.2218"))))

(time (dorun (map phone-numbers files)))
; "Elapsed time: 2649.807 msecs"
(time (dorun (pmap phone-numbers files)))
; "Elapsed time: 2772.794 msecs"
```

Единственное, что здесь изменилось – сами данные: каждая строка теперь имеет размер примерно 1 Кбайт. Даже при том, что общий объем данных остался прежним (здесь увеличилось количество «файлов»), накладные расходы на распараллеливание перекрыли преимущества, получаемые от распределения вычислений по разным потокам/ядрам. Из-за этих издержек прирост производительности при использовании `pmap` часто бывает меньше, чем в N раз (где N – количество доступных ядер процессора). Вывод очевиден: функцию `pmap` следует использовать в ситуациях, когда работа может выполняться параллельно, в несколько потоков, а затраты на обработку одного значения в последовательности существенно превосходят накладные расходы на координацию процесса, неизбежные при распараллеливании. Привлечение функции `pmap` для решения задач, не отвечающих этим условиям, может приводить к ухудшению производительности.

Однако в таких ситуациях часто существует обходное решение. Нередко можно эффективно распараллелить относительно простые операции, объединяя исходные данные в более крупные блоки. В примере выше текстовые блоки имеют размер всего 1 Кбайт. Однако размер обрабатываемого блока можно увеличить, например, каждое значение, обрабатываемое функцией `pmap`, можно представить в виде последовательности из 250 строк размером 1 Кбайт каждая и тем самым уменьшить накладные расходы:

```
(time (->> files
        (partition-all 250)
        (pmap (fn [chunk] (doall (map phone-numbers chunk))))  ❶
```

```
        (apply concat)
        dorun))
; "Elapsed time: 1465.138 msecs"
```

❶ map возвращает «ленивую» последовательность, поэтому здесь используется функция doall, обеспечивающая принудительную ее реализацию в области видимости функции, передаваемой функции pmap. В противном случае phone-numbers никогда не была бы вызвана в параллельных потоках выполнения из-за откладывания ее применения к строкам до момента обращения процесса к ленивой последовательности в будущем.

Изменив размер обрабатываемых блоков, удалось вернуть преимущества параллельной обработки, даже при том, что сложность вычислений возросла.

На функции pmap основаны две другие параллельные конструкции: pcalls и pvalues. Первая вызывает произвольное число функций без аргументов и возвращает «ленивую» последовательность их результатов. Вторая является макросом , который делает то же самое, но для выражений.

Состояние и идентичность

В языке Clojure *состояние* (state) и *идентичность* (identity) имеют ясные различия. Однако эти две концепции образуют неразделимый сплав, который во всей его полноте можно наблюдать в следующем примере:

```
class Person {
     public String name;
     public int age;
     public boolean wearsGlasses;

     public Person (String name, int age, boolean wearsGlasses) {
          this.name = name;
          this.age = age;
          this.wearsGlasses = wearsGlasses;
     }
}

Person sarah = new Person("Sarah", 25, false);
```

Вроде бы ничего необычного? Простой Java-класс[1] с несколькими полями, на основе которого можно создавать экземпляры. В действительности он имеет множество проблем.

Здесь мы создали ссылку на новый экземпляр класса `Person`, представляющий человека с именем `"Sarah"`, которому, по всей видимости, 25 лет от роду. С течением времени Сара (Sarah) проходила разные этапы развития (состояния): сначала она была ребенком, потом подростком и, наконец, стала взрослым человеком. В каждый момент времени – например, в прошлый вторник, в 11:07 – Сара находилась в каком-то *одном*, совершенно определенном состоянии, и каждое состояние с течением времени остается неизменным. Бессмысленно говорить об изменении любого из состояний, в котором пребывала Сара. Особенности прошлого вторника не изменятся в среду – она может переходить из одного состояния в другое, но это не изменит состояния, в котором она была прежде.

К сожалению, этот класс `Person` и низкоуровневые ссылки (в действительности обычные указатели), поддерживаемые большинством языков программирования, плохо подходят для представления даже такой тривиальной (можно даже сказать, фундаментальной) концепции. Когда Саре исполнится 26 лет, у нас остается только один способ отразить это – затереть определенное состояние, доступное нам[2]:

```
sarah.age++;
```

Еще более худшая ситуация складывается, когда возникает необходимость изменить сразу несколько признаков в состоянии Сары.

```
sarah.age++;
sarah.wearsGlasses = true;
```

В момент времени между выполнением этих двух строк кода оказывается, что возраст Сары увеличился, но она еще не носит очки. В течение некоторого времени (продолжительность которого, если

[1] Имейте в виду, что данное обсуждение не ограничивается рамками языка Java. Сплав состояния и идентичности можно найти во многих других языках (почти всех), включая Ruby, Python, C#, Perl, PHP и так далее.

[2] Пусть вас не смущает отсутствие методов доступа – описываемая семантика не зависит от того, как изменяется состояние, с помощью методов доступа или прямым обращением к полям.

говорить технически, зависит от особенностей процессора и среды выполнения), Сара может пребывать в противоречивом состоянии, которое семантически может считаться невозможным, в зависимости от объектной модели. Это благодатная почва для развития состояний гонки за ресурсами и взаимоблокировок.

Обратите внимание, что объект `sarah` можно изменить до такой степени, что он будет представлять совершенно другого человека:

```
sarah.name = "John";
```

Это доставляет беспокойство. Объект `sarah` не представляет ни единственного состояния Сары, ни даже понятия индивидуальности (или идентичности). Скорее, это безобразная смесь того и другого. Если говорить в более общем смысле, мы не можем утверждать что-либо о прежних состояниях ссылки на экземпляр класса `Person`. Каждый конкретный экземпляр класса `Person` может измениться в любой момент времени (что особенно характерно для многопоточных программ), а кроме того, их легко перевести в противоречивое состояние.

Подход, используемый в Clojure. В действительности нам требуется возможность, которая позволила бы сказать, что Сара обладает идентичностью, представляющей ее, – не ее состояние в какой-то момент времени, а ее саму, как сущность, существующую во времени. Кроме того, нам нужна возможность, позволяющая утверждать, что идентичность может пребывать в определенном состоянии в некоторый момент времени, но переход из одного состояния в другое не изменяет историю. Вспомните раздел «О важности значений» в главе 2 и сравнение изменяемых объектов с неизменяемыми значениями. Казалось бы, такие характеристики состояния не только дают практические преимущества, но и семантически выглядят более правильными. В конце концов, помимо гарантий, что состояние некоторой идентичности никогда не окажется в противоречивом состоянии (которые способны дать неизменяемые значения), мы можем пожелать сослаться на Сару, какой она была в прошлый вторник или в прошлом году.

В отличие от большинства объектов, структуры данных в языке Clojure являются неизменяемыми. Это делает их идеальными для представления состояния:

```
(def sarah {:name "Sarah" :age 25 :wears-glasses? false})
;= #'user/sarah
```

Ассоциативный массив, сохраняемый в переменной `sarah`, это состояние Сары в некоторый момент времени. Поскольку ассоциативный массив является неизменяемой структурой данных, можно быть уверенными, что любой код, хранящий ссылку на этот ассоциативный массив, всегда сможет использовать ее, независимо от того, какие изменения были выполнены в других версиях состояния или в состоянии, на которое ссылается переменная (var). Сами переменные в языке Clojure относятся к категории *ссылочных типов* (reference types) и фактически являются контейнерами, поддерживающими семантику одновременного доступа и изменения, которые могут хранить любые значения и использоваться в качестве устойчивой идентичности. То есть, можно сказать, что Сара представлена переменной `sarah`, состояние которой может изменяться с течением времени и в соответствии с семантикой переменных.

Это лишь беглый обзор понятий идентичности и состояния в языке Clojure, а также взаимосвязей между ними во времени, как различающимися концепциями[1]. Оставшаяся часть этой главы будет посвящена исследованию механизмов, реализующих эти концепции. Здесь будут представлены четыре ссылочных типа с различными семантиками изменения состояния с течением времени. Наряду с неизменяемыми значениями, эти ссылочные типы и их семантика позволяют проектировать многопоточные программы, максимально использующие все увеличивающиеся возможности компьютеров, и избавляться от целых категорий ошибок и проблем, характерных для мира низкоуровневых потоков выполнения и блокировок.

Ссылочные типы

Идентичность в языке Clojure может быть представлена с помощью четырех ссылочных типов: переменных (`var`), ссылок (`ref`), агентов (`agent`) и атомов (`atom`). Все они существенно отличаются друг от друга, но сначала мы поговорим об общих чертах, объединяющих их.

На самом низком уровне ссылки – это всего лишь контейнеры, хранящие значения, которые могут изменяться определенными

---

[1] Рич Хикки (Rich Hickey) сделал в 2009 году доклад об идеях идентичности, состояния и времени, и как они отразились на архитектуре Clojure. Мы настоятельно рекомендуем просмотреть видеоролик с докладом: http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey.

функциями (разными для разных ссылочных типов), как показано на рис. 4.1.

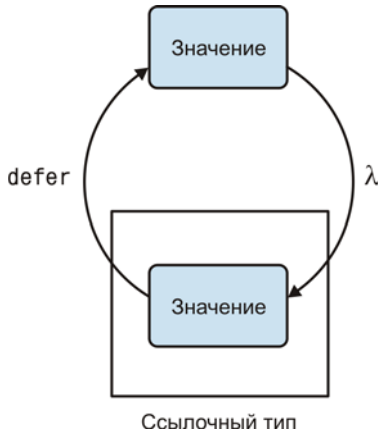

Рис. 4.1. Ссылочные типы в языке Clojure

Все ссылки всегда содержат *некоторые* значения (даже если это значение `nil`). Доступ к значениям всегда осуществляется с помощью `deref` или `@`:

```
@(atom 12)
;= 12
@(agent {:c 42})
;= {:c 42}
(map deref [(agent {:c 42}) (atom 12) (ref "http://clojure.org") (var +)])
;= ({:c 42} 12 "http://clojure.org"
;= #<core$_PLUS_clojure.core$_PLUS_@65297549>)
```

Операция разыменования возвращает *моментальный снимок* состояния ссылки на момент вызова `deref`. Это не означает, что при этом выполняется копирование значения. Вы просто получаете состояние, являющееся неизменным (здесь предполагается, что ссылка хранит неизменяемое значение, такое как коллекции в языке Clojure), но сама ссылка на состояние может изменяться с течением времени.

Одной из важнейших гарантий `deref` в контексте ссылочных типов является то обстоятельство, что `deref` *никогда не блокируется*, независимо от семантики разыменовываемого ссылочного типа или

операций, производимых в других потоках выполнения. Аналогично, операция разыменования ссылочного типа никогда не оказывает влияния на другие операции. Этим она резко отличается от механизмов `delay`, `promise` и `future` (которые могут заблокироваться на вызове `deref`, если значение отсутствует) и большинства параллельных структур данных в других языках, где нередко операция чтения может быть заблокирована операцией записи, и наоборот.

Операция «записи» значения в ссылочный тип имеет множество нюансов. Каждый ссылочный тип имеет свою семантику управления изменениями, и для каждого типа имеется отдельное семейство функций, выполняющих изменения в соответствии с семантикой. Обсуждение различных семантик и соответствующих им функций составит большую часть нашей дальнейшей дискуссии.

Помимо возможности разыменовывать их, все ссылочные типы:

- могут снабжаться метаданными (см. раздел «Метаданные» в главе 3); метаданные ссылочных типов могут изменяться только с помощью функции `alter-meta!`, изменяющей метаданные ссылки на месте[1];
- могут вызывать указанные вами функции при изменении состояния; эти функции называются *функциями-наблюдателями* (watches) и будут обсуждаться в разделе «Функции-наблюдатели» ниже;
- позволяют накладывать ограничения на хранимое состояние с помощью *функций-валидаторов* (validators) и предотвращать нежелательные изменения (см. раздел «Функции проверки» ниже).

Классификация параллельных операций

В процессе обсуждения ссылочных типов, мы постоянно будем сталкиваться с двумя ключевыми понятиями, используемыми для описания параллельных операций. Вместе взятые, они могут помочь прояснить, когда лучше использовать тот или иной тип.

Координация. *Скоординированной* называется операция, обеспечивающая согласованность действий с множеством объектов (или,

---

[1] Функции `atom`, `ref` и `agent` принимают дополнительный именованный аргумент `:meta`, в котором передается начальный ассоциативный массив с метаданными.

как минимум, их изолированность друг от друга) и получение правильных результатов. Классическим примером таких операций являются банковские транзакции: процесс, выполняющий перевод денег с одного счета на другой должен гарантировать, что сумма на счете получателя не будет увеличена до уменьшения суммы на счете отправителя, и что вся транзакция потерпит неудачу, если на счете отправителя окажется недостаточно средств. Одновременно подобные операции с этими же счетами могут производить многие другие процессы. В отсутствие механизмов координации изменений некоторые счета иногда могли бы отражать неправильный баланс, а транзакции, которые должны потерпеть неудачу (или завершиться успехом), могли бы по ошибке успешно завершаться (или терпеть неудачу).

Напротив, *нескоординированной* называется операция, в которой участвующие объекты не могут оказывать отрицательного влияния друг на друга. Например, два различных потока выполнения могут безопасно выполнять запись в два разных файла, не влияя друг на друга.

Синхронизация. *Синхронной* называется операция, которая блокирует вызывающий поток выполнения до момента, пока он не сможет получить исключительный доступ к требуемому контексту, тогда как *асинхронной* называется операция, которая может быть выполнена без блокирования вызывающего потока выполнения.

Рис. 4.2. Классификация ссылочных типов в языке Clojure

Этих двух понятий (или четырех, если принять во внимание их зеркальные отражения) достаточно, чтобы полностью охарактеризовать многие (если не большинство) параллельные операции, с которыми вы можете столкнуться. С учетом этого в языке Clojure было реализовано четыре ссылочных типа (рис. 4.2), отражающих семантику различных сочетаний этих понятий, которые удачно классифицируются по типам операций[1].

[1] Переменные (vars) не укладываются в эту классификацию. При изменении, их значения становятся локальными в области видимости потоков выполнения, поэтому они ортогональны понятиям координации и синхронизации.

Помните об этой классификации при выборе ссылочного типа для решения той или иной задачи; если вы сможете охарактеризовать задачу с ее использованием, тогда выбор ссылочного типа будет очевиден.

Примечание. Вы могли заметить отсутствие ссылочного типа, соответствующего скоординированной асинхронной семантике. Эта комбинация характеристик наиболее типична для распределенных систем, таких как базы данных, гарантирующих объединение изменений в единую модель. Главной же целью Clojure, напротив, является поддержка конкуренции и параллелизма внутри процесса.

Демонстрационные утилиты

Для более наглядной демонстрации конкуренции, в примерах этой главы будут использоваться вспомогательные макросы. `futures` – макрос, создающий `n` объектов `future` для вычисления выражения, переданного макросу:

```
(defmacro futures
  [n & exprs]
  (vec (for [_ (range n)
             expr exprs]
         `(future ~expr))))
```

Он позволяет легко организовать вычисление выражений в разных потоках выполнения. Сам макрос `futures` возвращает вектор с созданными объектами `future`. Это может пригодиться в других контекстах, но мы всегда будем ждать, пока все объекты `future` не завершат выполнение, чтобы быть уверенными, что все выражения вычислены. Этой цели служит другой вспомогательный макрос, `waitfutures`, предоставляющий те же возможности, что и `futures`, но всегда возвращающий `nil` и блокирующий REPL до завершения всех вычислений:

```
(defmacro wait-futures
  [& args]
  `(doseq [f# (futures ~@args)]
     @f#))
```

Мы еще не обсуждали макросы, поэтому нет ничего страшного, если вам непонятно, как они действуют – подробно о макросах будет рассказываться в главе 5.

Атомы

Атомы – это самый простой ссылочный тип. Они являются идентичностями, реализующими синхронную нескоординированную семантику, атомарность изменений, выполняемых по принципу «сравнить и изменить» (compare-and-set). То есть, операции, изменяющие состояния атомов, будут блокировать вызывающую программу до завершения изменений, и каждое изменение будет выполняться изолированно – в Clojure нет возможности организовать одновременное изменение двух атомов.

Создаются атомы с помощью функции `atom`. Функция `swap!` представляет наиболее часто используемую операцию изменения атомов, которая записывает в атом результат применения к значению атома некоторой функции с дополнительными аргументами, переданными функции `swap!`:

```
(def sarah (atom {:name "Sarah" :age 25 :wears-glasses? false}))
;= #'user/sarah
(swap! sarah update-in [:age] + 3)                              ❶
;= {:age 28, :wears-glasses? false, :name "Sarah"}              ❷
```

❶ Здесь функция `swap!` запишет в атом `sarah` результат вызова `(update-in @sarah [:age] + 3)`.

❷ `swap!` всегда возвращает новое значение, помещенное в атом.

Атомы – это минимум из того, что можно использовать для корректного изменения состояния Сары: каждое изменение атома выполняется атомарно, поэтому к значению атома можно смело применять любые функции или композиции функций и быть уверенными, что никакой другой поток выполнения никогда не получит содержимое атома в противоречивом или частично измененном состоянии:

```
(swap! sarah (comp #(update-in % [:age] inc)
                   #(assoc % :wears-glasses? true)))
;= {:age 29, :wears-glasses? true, :name "Sarah"}
```

Используя функцию `swap!` следует помнить, что из-за поддержки семантики «сравнить и изменить», если значение атома изменится (например, в ходе выполнения операций в другом потоке) до того, как функция `update` вернет результат, `swap!` повторит попытку, вызвав `update` с обновленным значением атома. Такие попытки будут повторяться, пока функция `swap!` не преуспеет:

```
(def xs (atom #{1 2 3}))
;= #'user/xs
(wait-futures 1 (swap! xs (fn [v]
                            (Thread/sleep 250)
                            (println "trying 4")
                            (conj v 4)))
                (swap! xs (fn [v]
                            (Thread/sleep 500)
                            (println "trying 5")
                            (conj v 5))))
;= nil
; trying 4
; trying 5      ❶
; trying 5
@xs
;= #{1 2 3 4 5}
```

❶ Поток выполнения, добавляющий 5 в множество xs, вынужден повторить попытку применения функции, переданной функции swap!; пока этот поток простаивал, другой поток изменил состояние атома (добавив 4 в множество), из-за чего первая попытка «сравнить и изменить» потерпела неудачу.

Семантику функции swap! можно изобразить, как показано на рис. 4.3.

Если значение атома a изменится между моментом вызова функции g и моментом, когда она вернет новое значение для a (a1 и a2,

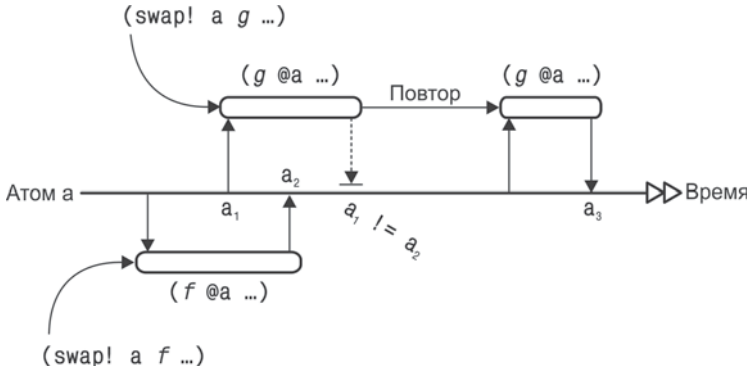

Рис. 4.3. Выполнение функцией swap! конфликтующих операций с атомом

соответственно), swap! отбросит новое значение и повторит вызов для изменившегося значения a. Попытки будут продолжаться, пока значение, полученное в результате вызова g, не окажется вычисленным на основе текущего значения a.

В Clojure нет механизмов, ограничивающих семантику повторных попыток функции swap!; учитывая это обстоятельство, функция, передаваемая swap!, *должна* быть чистой, иначе последствия могут оказаться самыми непредсказуемыми.

Так как атомы являются синхронными ссылочными типами, функции, изменяющие значения атомов не возвращают управление, пока операция не будет выполнена полностью:

```
(def x (atom 2000))
;= #'user/x
(swap! x #(Thread/sleep %))     ❶
;= nil
```

❶ Это выражение вернет управление не ранее, чем через две секунды.

К атомам можно также применять простую операцию compare-and-set!, если текущее значение атома уже известно – она возвращает true, только если значение атома изменилось:

```
(compare-and-set! xs :wrong "new value")
;= false
(compare-and-set! xs @xs "new value")
;= true
@xs
;= "new value"
```

Внимание. Функция compare-and-set! не использует семантику значения – она требует, чтобы значение атома было идентично[1] ожидаемому, которое передается во втором аргументе:

```
(def xs (atom #{1 2}))
;= #'user/xs
(compare-and-set! xs #{1 2} "new value")
;= false
```

[1] Как определяется предикатом identical? – см. раздел «Идентичность объектов (identical?)» в главе 11.

Наконец, существует «атомный инструмент»: если потребуется изменить состояние атома без учета его текущего содержимого, используйте `reset!`:

```
(reset! xs :y)
;= :y
@xs
;= :y
```

Теперь, когда вы познакомились с атомами, самое время взглянуть на две возможности, которыми обладают все ссылочные типы, поскольку они будут использоваться в некоторых примерах ниже.

Уведомление и ограничение

В разделе «Ссылочные типы» мы уже познакомились с одной из наиболее часто используемых операций – разыменованием – возвращающей текущее значение ссылки, независимо от ее типа. Однако существуют и другие операции, общие для всех ссылочных типов, которые иногда могут потребоваться для слежения за изменениями и проверки новых значений. Все ссылочные типы в языке Clojure предоставляют такую возможность посредством регистрации *функций-наблюдателей* и *функций-валидаторов*.

Функции-наблюдатели

Функции-наблюдатели вызываются при каждом изменении состояния ссылки. Если вы знакомы с шаблоном проектирования «Наблюдатель» (Observer), вы сразу сможете понять назначение этих функций, даже при том, что функции-наблюдатели представляют собой более универсальный инструмент: функция-наблюдатель может быть зарегистрирована в любом ссылочном типе и все функции-наблюдатели являются обычными функциями – от них не требуется реализовать какой-то определенный интерфейс.

Сразу после создания ссылочные типы не имеют функций-наблюдателей, но они могут подключаться и отключаться в любой момент. Функция-наблюдатель должна принимать четыре аргумента: ключ, ссылку, претерпевшую изменение (`atom`, `ref`, `agent` или `var`), прежнее состояние ссылки и новое состояние:

```
(defn echo-watch                                                    ❶
  [key identity old new]
  (println key old "=>" new))
;= #'user/echo-watch
(def sarah (atom {:name "Sarah" :age 25}))
;= #'user/sarah
(add-watch sarah :echo echo-watch)
;= #<Atom@418bbf55: {:name "Sarah", :age 25}>
(swap! sarah update-in [:age] inc)
; :echo {:name Sarah, :age 25} => {:name Sarah, :age 26}
;= {:name "Sarah", :age 26}
(add-watch sarah :echo2 echo-watch)                                 ❷
;= #<Atom@418bbf55: {:name "Sarah", :age 26}>
(swap! sarah update-in [:age] inc)                                  ❸
; :echo {:name Sarah, :age 26} => {:name Sarah, :age 27}
; :echo2 {:name Sarah, :age 26} => {:name Sarah, :age 27}
;= {:name "Sarah", :age 27}
```

❶ Наша функция-наблюдатель производит вывод в `stdout` при каждом изменении состояния атома.
❷ Если эту же функцию подключить с другим ключом...
❸ ...она будет вызываться дважды при каждом изменении состояния.

> **Внимание.** Функции-наблюдатели вызываются синхронно, в том же потоке выполнения, где изменяется состояние ссылки. Это означает, что к моменту вызова функции-наблюдателя, состояние ссылки может быть изменено в другом потоке выполнения. То есть, при выполнении операций следует полагаться на «старое» и «новое» значения, передаваемые функции-наблюдателю, вместо того, чтобы пытаться разыменовать ссылку.

Ключ, указанный в вызове `add-watch`, можно использовать для отключения функции-наблюдателя:

```
(remove-watch sarah :echo2)
;= #<Atom@418bbf55: {:name "Sarah", :age 27}>
(swap! sarah update-in [:age] inc)
; :echo {:name Sarah, :age 27} => {:name Sarah, :age 28}
;= {:name "Sarah", :age 28}
```

Обратите внимание, что функции-наблюдатели вызываются при попытках изменить состояние ссылки, но это не означает, что состояние стало *другим*:

```
(reset! sarah @sarah)
; :echo {:name Sarah, :age 28} => {:name Sarah, :age 28}
;= {:name "Sarah", :age 28}
```

По этой причине перед выполнением каких-либо операций в функциях-наблюдателях принято проверять равенство прежнего и нового состояний ссылки.

Вообще говоря, функции-наблюдатели являются отличным механизмом распространения локальных изменений по другим ссылкам или системам. Например, они делают чрезвычайно простой реализацию механизма журналирования изменений состояния ссылки:

```
(def history (atom ()))

(defn log->list
  [dest-atom key source old new]
  (when (not= old new)
    (swap! dest-atom conj new)))

(def sarah (atom {:name "Sarah", :age 25}))
;= #'user/sarah
(add-watch sarah :record (partial log->list history))    ❶
;= #<Atom@5143f787: {:age 25, :name "Sarah"}>
(swap! sarah update-in [:age] inc)
;= {:age 26, :name "Sarah"}
(swap! sarah update-in [:age] inc)
;= {:age 27, :name "Sarah"}
(swap! sarah identity)                                   ❷
;= {:age 27, :name "Sarah"}
(swap! sarah assoc :wears-glasses? true)
;= {:age 27, :wears-glasses? true, :name "Sarah"}
(swap! sarah update-in [:age] inc)
;= {:age 28, :wears-glasses? true, :name "Sarah"}
(pprint @history)
;= ;= nil
;= ; ({:age 28, :wears-glasses? true, :name "Sarah"}
;= ; {:age 27, :wears-glasses? true, :name "Sarah"}
;= ; {:age 27, :name "Sarah"}
;= ; {:age 26, :name "Sarah"})
```

❶ Здесь функция `partial` используется для связывания с атомом, куда будет сохраняться история изменений.

❷ Поскольку `identity` всегда возвращает свой аргумент в неизменном виде, этот вызов `swap!` произведет эффект изменения состояния ссылки, но старое и новое состояния будут равны. Функция `log->list` добавляет запись в журнал, только если новое состояние отличается, поэтому такие попытки не сохраняются в истории.

При необходимости можно даже реализовать разное поведение функции-наблюдателя, зависящее от ключа, под которым она зарегистрирована. Простейшим примером может служить функция, сохраняющая информацию об изменениях не в памяти, а в базе данных, определяемой ключом:

```
(defn log->db
  [db-id identity old new]
  (when (not= old new)
    (let [db-connection (get-connection db-id)]
      ...)))

(add-watch sarah "jdbc:postgresql://hostname/some_database" log->db)
```

В разделе «Сохранение состояний ссылок в журнале на основе агента», ниже, для большего эффекта мы объединим функции-наблюдатели со ссылками и агентами.

Функции-валидаторы

Валидаторы (validators) позволяют накладывать собственные ограничения на изменение состояния. Валидатор – это функция с единственным аргументом, вызываемая непосредственно перед записью *предлагаемого* состояния в ссылку. Если валидатор вернет логически ложное значение или возбудит исключение, операция изменения состояния будет прервана с исключением.

Предлагаемое изменение является результатом действия любых функций изменения состояния, которые могут применяться к ссылке. Например, после того, как будет увеличено значение слота `:age` ассоциативного массива в ссылке `sarah`, но перед тем, как `swap!` запишет изменившееся состояние в ссылку. Это как раз тот момент, когда функция-валидатор – если она была зарегистрирована для данной ссылки – получит шанс наложить свое вето.

```
(def n (atom 1 :validator pos?))
;= #'user/n
```

```
(swap! n + 500)
;= 501
(swap! n - 1000)
;= #<IllegalStateException java.lang.IllegalStateException: Invalid
;= reference state>
```

Функции-валидаторы принимают единственный аргумент, поэтому, поэтому в данном качестве можно использовать любые подходящие предикаты, такие как `pos?`.

Зарегистрировать валидатор можно для любого ссылочного типа, однако для атомов, ссылок и агентов валидаторы можно указывать непосредственно *при создании*, в параметре `:validator` функций `atom`, `ref` и `agent`. Добавить валидатор в переменную или сменить валидатор, подключенный к атому, ссылке или агенту, можно с помощью функции `set-validator!`:

```
(def sarah (atom {:name "Sarah" :age 25}))
;= #'user/sarah
(set-validator! sarah :age)
;= nil
(swap! sarah dissoc :age)
;= #<IllegalStateException java.lang.IllegalStateException: Invalid
;= reference state>
```

Сообщение, включаемое в исключение, можно сделать информативнее, если в случае ошибки при проверке валидатор будет не просто возвращать `false` или `nil`, а возбуждать собственные исключения[1]:

```
(set-validator! sarah #(or (:age %)      ❶
                           (throw (IllegalStateException.
                                    "People must have `:age`s!"))))
;= nil
(swap! sarah dissoc :age)
;= #<IllegalStateException java.lang.IllegalStateException:
;=  People must have `:age`s!>
```

❶ Не забывайте, что валидаторы должны возвращать логически истинное значение, иначе на изменение будет наложено вето. В данном случае, если бы мы реализовали валидатор, например, в виде: `#(when-`

[1] Можно также использовать библиотеку, такую как Slingshot, позволяющую возбуждать *значения*, вместо того, чтобы пытаться представить полезную информацию в строке: https://github.com/scgilardi/slingshot.

`not (:age %) (throw ...))`, он возвращал бы `nil` для состояний, обладающих слотом `:age`, вызывая непреднамеренную ошибку проверки.

Валидаторы имеют большую практическую ценность в общем случае, однако они имеют особый статус в отношении ссылок, о чем будет рассказываться далее, а точнее в разделе «Обеспечение локальной целостности с применением валидаторов».

Ссылки

Ссылки (`ref`) – это скоординированный ссылочный тип. При использовании ссылок в параллельных операциях с несколькими идентичностями:

- исключается возможность выполнения параллельных операций со ссылками, находящимися в противоречивом состоянии;
- исключаются ситуации гонки за ресурсами;
- отсутствует необходимость использовать блокировки, мониторы и другие низкоуровневые механизмы синхронизации;
- исключаются взаимоблокировки.

Все это возможно, благодаря поддержке *программной транзакционной памяти* (Software Transactional Memory, STM) в Clojure, используемой для управления всеми изменениями, применяемыми к значениям, хранящимся в ссылках.

Программная транзакционная память

Если говорить в общих чертах, программная транзакционная память (STM) – это способ координации конкурирующих изменений в совместно используемых блоках памяти. Практически во всех языках программирования это означает необходимость вручную управлять блокировками. Поддержка STM предлагает другую альтернативу.

Так же как автоматическая сборка мусора в значительной степени избавила от ручного управления распределением памяти – устранив широкий диапазон трудноуловимых (и не очень) ошибок, связанных с этим, – программная транзакционная память, часто характеризуемая как аналогичное средство, устраняющее другую категорию программных ошибок, избавляет от ручного управления блокировками. В обоих случаях использование проверенного автоматизированного решения, снижающего вероятность ошибок, одновременно освобож-

дает от необходимости отвлекаться на низкоуровневые операции, не связанные с предметной областью, и часто позволяет в конечном результате получить более высокие эксплуатационные характеристики, чем низкоуровневые решения[1].

Программная транзакционная память в Clojure реализована с использованием приемов, десятилетиями оттачивавшихся в системах управления базами[2]. Как можно заключить из названия, каждое изменение множества ссылок наделено семантикой транзакций, хорошо знакомой тем, кому приходилось работать с базами данных. Каждая транзакция гарантирует, что изменения в ссылках будут выполнены:

1. *Атомарно*, то есть, либо будут применены все изменения, либо ни одно.
2. *Согласованно*, то есть, транзакция потерпит неудачу, если изменения в каких-либо ссылках не будут соответствовать наложенным ограничениям.
3. *Изолированно*, то есть, транзакция не будет оказывать влияния на состояния ссылок, наблюдаемые изнутри других транзакций или потоков выполнения.

Таким образом, поддержка программной транзакционной памяти в Clojure удовлетворяет составляющим «A», «C» и «I» аббревиатуры ACID (https://ru.wikipedia.org/wiki/ACID), известной в мире баз данных. Отсутствие составляющей «D» (Durability – надежность[3]) нельзя назвать недостатком, так как механизм транзакционной памяти использует обычную оперативную память компьютера[4].

---

[1] Применение современных реализаций механизмов сборки мусора во многих случаях позволяет получить выигрыш, в сравнении с использованием приемов управления памятью вручную; и каждый раз, когда в JVM добавляется новая реализация или оптимизация сборщика мусора, все программы автоматически получают дополнительные выгоды от этого без участия их создателей. Та же динамика наблюдается и в случае с программной транзакционной памятью в Clojure.

[2] В частности, многовариантный контроль совпадений (Multiversion Concurrency Control, MVCC): https://en.wikipedia.org/wiki/Multiversion_concurrency_control.

[3] Здесь имеется в виду надежность, или долговременность хранения. – *Прим. перев.*

[4] Пример обеспечения надежности хранения состояния ссылки будет представлен в разделе «Сохранение состояний ссылок в журнале на основе агента», ниже.

Механика изменения ссылок

Теперь можно перейти к знакомству с особенностями ссылок. Выше, в разделе «Классификация параллельных операций», упоминались банковские операции, как пример операций, требующих координации изменений среди нескольких идентичностей и потоков выполнения. Однако, демонстрация семантики транзакций на примере банковских операций может выглядеть слишком сложной. Пожалуй, более поучительным (и интересным) будет выглядеть пример использования ссылок и программной транзакционной памяти в игровом движке, поддерживающем участие в игре сразу нескольких игроков.

В то время как некоторые задачи называются «естественно параллельными» (embarrassingly parallel) из-за простоты их разделения на параллельные операции, задачу поддержки нескольких игроков можно назвать «естественно конкурентной» (embarrassingly concurrent): она часто вовлекает в операции массивные наборы данных, и в игре могут участвовать сотни и тысячи независимых игроков, вызывающих изменения, которые должны применяться скоординировано и согласованно, чтобы обеспечить соблюдение правил игры.

Наша «игра»[1] будет написана в жанре фантастических ролевых игр с такими персонажами, как колдуны, странники и барды. Каждый персонаж будет представлен в виде ссылки, хранящей ассоциативный массив со всеми данными и способностями. Независимо от принадлежности к тому или иному классу, все персонажи будут иметь минимальный набор атрибутов:

- `:name`, имя персонажа в игре;
- `:health`, число, определяющее физическое состояние персонажа. Когда значение `:health` снижается до `0`, персонаж погибает;
- `:items`, множество предметов, которые несет персонаж.

Разумеется, разные классы персонажей будут иметь собственные атрибуты. `character` – это функция, реализующая все это, со значениями `:items` и `:health` по умолчанию:

```
(defn character
  [name & {:as opts}]
  (ref (merge {:name name :items #{} :health 500}
              opts)))
```

[1] Мы не занимаемся разработкой игр, и то, что будет создаваться здесь, является лишь упрощенным примером, однако демонстрируемые механизмы вполне можно использовать для реализации полноценного игрового движка.

Теперь с помощью этой функции можно определить несколько персонажей, которыми могли бы управлять разные игроки[1]:

```
(def smaug (character
              "Smaug" :health 500 :strength 400 :items (set (range 50))))    ❶
(def bilbo (character "Bilbo" :health 100 :strength 100))
(def gandalf (character "Gandalf" :health 75 :mana 750))
```

❶ Персонаж `smaug` создается с дополнительным комплектом предметов; здесь предметы представлены целыми числами, которые могут соответствовать числовым идентификаторам предметов в статическом ассоциативном массиве или внешней базе данных.

Если бы в игре, подобной этой, Бильбо (Bilbo) и Гендальф (Gandalf) смогли победить в битве со Смогом (Smaug), они могли бы забрать «имущество» Смога. Не вдаваясь в детали игры, это означает, что нам требуется организовать передачу предметов, которые носил Смог, другим персонажам. Передача должна осуществляться так, чтобы с точки зрения внешнего наблюдателя передаваемый предмет в каждый конкретный момент времени находился только в одном месте.

Введем в дело поддержку транзакционной памяти и транзакции. Функция `dosync` открывает транзакцию[2]. *Все изменения в ссылке должны производиться только внутри транзакции*, которая обрабатывается синхронно. То есть, поток выполнения, инициировавший транзакцию, «заблокируется» до завершения этой транзакции и только потом продолжит работу.

По аналогии с функцией `swap!`, если две транзакции попытаются выполнить конфликтующие изменения в одной или более ссылках,

---

[1] В настоящем игровом движке едва ли кто-то будет использовать отдельные переменные для хранения персонажей. Гораздо удобнее хранить всех активных персонажей в едином ассоциативном массиве, который сам хранится в ссылке. Когда игроки будут подключаться к игре или выходить из нее, их персонажи будут добваляться в ассоциативный массив или удаляться из него.

[2] Обратите внимание, что вложенные области видимости транзакций – либо лексически вложенные формы `dosync`, либо образованные различными функциями, встречающимися на пути потока выполнения – объединяются в единую логическую транзакцию, которая подтверждается или отменяется как единое целое, когда поток выполнения выходит за пределы самой внешней формы `dosync`.

одна из транзакций вынуждена будет повторить попытку. Являются ли две конкурирующие транзакции конфликтующими, определяют функции, применяемые для изменения ссылок, совместно используемых в этих транзакциях. Всего имеется три таких функции – `alter`, `commute` и `ref-set` – с разной семантикой, с точки зрения определения состояния конфликта.

Так как же нам реализовать передачу предметов между персонажами? Функция `loot` выполняет передачу одного значения из `(:items @from)` в `(:items @to)` в рамках транзакции, предполагая, что оба являются множествами[1], и возвращает новое состояние `from`:

Пример 4.2. Передача предметов

```
(defn loot
  [from to]
  (dosync
    (when-let [item (first (:items @from))]          ❶
      (alter to update-in [:items] conj item)
      (alter from update-in [:items] disj item))))
```

❶ Если множество `(:items @from)` окажется пустым, функция `first` вернет `nil`, тело `when-let` не будет выполнено, в рамках транзакции не будет выполнено никаких действий и сама функция `loot` вернет `nil`.

Предположим, что Смог побежден и мы можем заставить Бильбо и Гендальфа забрать его имущество:

```
(wait-futures 1
              (while (loot smaug bilbo))
              (while (loot smaug gandalf)))
;= nil
@smaug
;= {:name "Smaug", :items #{}, :health 500}
@bilbo
;= {:name "Bilbo", :items #{0 44 36 13 ... 16}, :health 500}
@gandalf
;= {:name "Gandalf", :items #{32 4 26 ... 15}, :health 500}
```

Теперь Гендальф и Бильбо забрали предметы, имевшиеся у Смога. Важно отметить, что персонажи `bilbo` и `gandalf` поделили добы-

[1] В разделе «Множества», в главе 3, уже говорилось, что `disj` возвращает множество, не содержащее данное значение.

чу с помощью механизма `future` (то есть, потоков выполнения), и передача каждого предмета произошла атомарно: ни один предмет не был потерян, ни один предмет не был продублирован и ни один предмет не был переда сразу нескольким персонажам.

Пример 4.3. Проверка согласованности результатов выполнения функции loot

```
(map (comp count :items deref) [bilbo gandalf])      ❶
;= (21 29)
(filter (:items @bilbo) (:items @gandalf))           ❷
;= ()
```

❶ Если бы сумма этих счетчиков отличалась от `50` (число предметов, имевшихся у Смога) или...

❷ ...если бы Гендальф получил те же предметы, что и Бильбо, тогда результат наших транзакций передачи предметов оказался бы противоречивым.

Нам удалось избежать ручного управления блокировками и создать масштабируемую реализацию, которую легко адаптировать для работы с большим количеством ссылок и большим количеством транзакций, выполняемых в отдельных потоках.

Функция alter

Функция `loot` использует функцию `alter`, которая подобно функции `swap!` принимает ссылку, некоторую функцию `f`, и дополнительные аргументы для нее. По возвращении из функции `alter`, в ссылку будет записано *значение, полученное в транзакции*, возвращаемое функцией `f`, которая получает первый аргумент со значением ссылки и все дополнительные аргументы, переданные функции `alter`.

Понятие значения, полученного в транзакции, играет важную роль. Все функции, изменяющие состояние ссылки, в действительности работают в гипотетической шкале времени жизни состояния ссылки, начинающейся с момента *первого изменения*. Все остальные обращения к значению ссылки и его изменения производятся в этой отдельной шкале, существующей и доступной только в рамках транзакции. Когда поток управления выходит из транзакции, механизм транзакционной памяти пытается подтвердить изменения. В случае успеха все значения, полученные в транзакции для каждой ссылки, будут сохранены в ссылках и станут доступны остальной программе. Однако, в зависимости от семантики операций, используемых для

вычисления значений в транзакции, любые изменения состояний ссылок, выполненные за пределами транзакции, *могут* конфликтовать с изменениями, произведенными внутри транзакции, по этой причине транзакцию будет перезапущена повторно, с самого начала.

В течение этого процесса, любой поток выполнения, производящий только *чтение* (то есть, разыменование) ссылки, вовлеченной в транзакцию, сможет получить необходимое значение без приостановки на блокировке. Кроме того, пока данная транзакция не будет подтверждена, изменения, произведенные в ней, не будут доступны читающему коду за пределами транзакции, включая и читающий код, выполняющийся в рамках другой транзакции.

Уникальность семантики функции `alter` состоит в том, что до подтверждения транзакции, значение ссылки за пределами транзакции должно оставаться прежним, каковым оно было до первого применения функции `alter` внутри транзакции. В противном случае транзакция будет перезапущена с новым значением ссылки.

Этот процесс можно изобразить на примере взаимодействия двух транзакций, `t1` и `t2`, воздействующих с помощью функции `alter` на одну и ту же ссылку `a` (см. рис. 4.4).

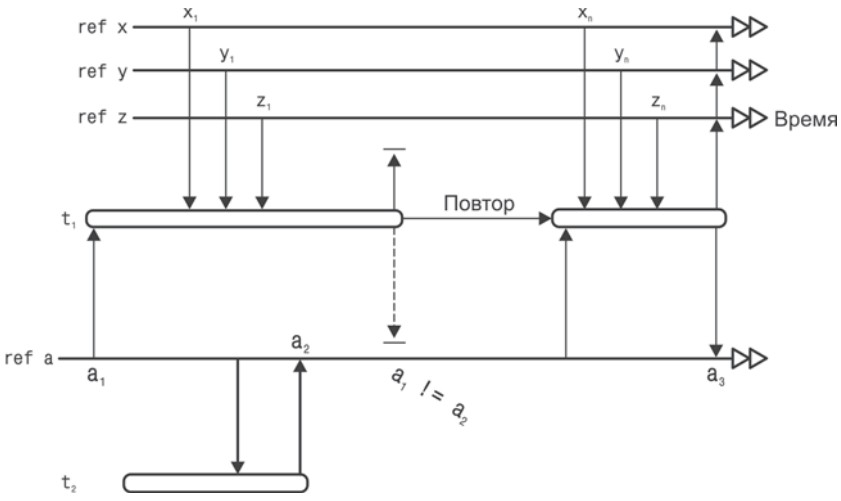

Рис. 4.4. Взаимодействие транзакций, применяющих конфликтующие изменения к одной и той же ссылке с помощью функции alter

Даже при том, что транзакция `t1` запускается первой, попытка подтвердить ее оканчивается неудачей, потому что за время ее работы, транзакция `t2` успела изменить ссылку: текущее состояние ссылки (`a2`) отличается от состояния (`a1`), которое существовало на момент запуска транзакции `t1`. Этот конфликт отменяет все изменения, произведенные в транзакции `t1` (например, `x`, `y`, `z`, ...). После этого транзакция `t1` перезапускается, получая обновленные значения всех используемых в ней ссылок.

Согласно схеме и описанию, представленным выше, механизм транзакционной памяти в языке Clojure можно представлять как процесс, который оптимистично переупорядочивает конкурирующие операции так, что они выполняются последовательно. Неудивительно, что ту же самую семантику можно обнаружить в мире баз данных, под названием *последовательная изоляция моментального снимка* (Serializable Snapshot Isolation, SSI) (https://en.wikipedia.org/wiki/Serializability).

> **Внимание.** Результаты транзакции не будут подтверждены, если к моменту подтверждения обнаружатся какие-либо конфликты. То есть, достаточно обнаружить конфликт в единственной ссылке, чтобы транзакция перезапустилась, даже если изменение сотни других ссылок можно было бы смело подтвердить.

Уменьшение конфликтов в транзакциях с помощью commute

Поскольку относительно порядка следования изменений не делается никаких предположений, функция `alter` является самым безопасным механизмом для воздействия на ссылки. Однако бывают ситуации, когда операции не могут применяться в каком-то другом порядке. В таких случаях вместо `alter` можно использовать `commute`, уменьшающую вероятность конфликтов и повторных попыток, и тем самым увеличивающую общую пропускную способность.

Как можно догадаться из имени, `commute` относится к *коммутативным* (commutative) функциям (https://en.wikipedia.org/wiki/Commutative_property) – функциям, чей результат не изменяется от перестановки аргументов, таким как `+`, `*`, `clojure.set/union`... – но не требует, чтобы передаваемые ей функции в свою очередь были коммутативными. Единственное, что требует `commute`, чтобы семантика программы не зависела от возможного изменения порядка применения передаваемых ей функций. Из этого следует, что в таких

случаях значение имеет только конечный результат применения всех функций, а не их промежуточные результаты.

Например, несмотря на то, что операция деления не является коммутативной, ее часто можно использовать с функцией `commute`, когда промежуточные результаты не имеют значения:

```
(= (/ (/ 120 3) 4) (/ (/ 120 4) 3))
;= true
```

Таким образом, `commute` можно использовать везде, где композиция вовлекаемых функций коммутативна:

```
(= ((comp #(/ % 3) #(/ % 4)) 120) ((comp #(/ % 4) #(/ % 3)) 120))
;= true
```

В общем случае функцию `commute` следует использовать для применения изменений к состояниям ссылок, только когда порядок этих изменений не имеет значения.

Функция `commute` имеет два важных отличия от `alter`. Во-первых, значение, возвращаемое функцией `alter`, будет записано в ссылку при подтверждении транзакции. Иными словами, значение, полученное в транзакции, является значением, которое возможно будет подтверждено. Значение, полученное в транзакции функцией `commute`, напротив, не обязательно станет подтверждаемым значением, потому что коммутируемая функция (переданная функции `commute`) будет применена снова на этапе подтверждения с последним значением коммутируемой ссылки.

Во-вторых, изменение значений ссылок с помощью `commute` *никогда* не вызывает конфликтов и потому никогда не приводит к повторному выполнению транзакций. Очевидно, что это имеет важные последствия для производительности и пропускной способности: повторные попытки выполнения транзакций требуют дополнительного времени и «блокируют» выполнение потока до успешного завершения транзакции, отодвигая во времени переход к выполнению следующей задачи.

Это легко продемонстрировать на примере. Пусть имеется некоторая ссылка `x`:

```
(def x (ref 0))
;= #'user/x
```

Попробуем выполнить с ней 10 000 транзакций, каждая из которых выполняет небольшой объем работы (просто получает сумму нескольких целых чисел) и затем изменяет значение x:

```
(time (wait-futures 5
                    (dotimes [_ 1000]
                      (dosync (alter x + (apply + (range 1000)))))
                    (dotimes [_ 1000]
                      (dosync (alter x - (apply + (range 1000)))))))
; "Elapsed time: 1466.621 msecs"
```

Некоторая часть этого времени была потрачена на повторные выполнения транзакций и, соответственно, на повторные вычисления сумм последовательностей целых чисел. Однако операции, выполняемые здесь с помощью alter (сложение и вычитание) с тем же успехом можно выполнять и с помощью commute:

```
(time (wait-futures 5
                    (dotimes [_ 1000]
                      (dosync (commute x + (apply + (range 1000)))))
                    (dotimes [_ 1000]
                      (dosync (commute x - (apply + (range 1000)))))))
; "Elapsed time: 818.41 msecs"
```

Даже при том, что она применяет функцию изменения к значению ссылки дважды – один раз, чтобы получить значение в транзакции (чтобы ссылка x получила новое значение, которое можно было бы использовать далее в этой же транзакции), и второй раз на этапе подтверждения, чтобы выполнить «фактическое» изменение значения x (возможно уже изменившееся) – общее время выполнение уменьшилось почти наполовину, благодаря тому, что commute никогда не выполняет повторы транзакций.

Однако в commute нет никакого волшебства: ее следует использовать с осторожностью, иначе можно получить ошибочные результаты. Посмотрим, что произойдет, если по невнимательности вместо alter использовать commute в функции loot из примера 4.2:

Пример 4.4. Некорректная функция передачи предметов, использующая commute

```
(defn flawed-loot
  [from to]
  (dosync
```

```
    (when-let [item (first (:items @from))]
    (commute to update-in [:items] conj item)
    (commute from update-in [:items] disj item))))
```

Создадим наших персонажей заново и посмотрим, как работает новая функция `flawed-loot`:

```
(def smaug (character "Smaug" :health 500 :strength 400 :items (set (range 50))))
(def bilbo (character "Bilbo" :health 100 :strength 100))
(def gandalf (character "Gandalf" :health 75 :mana 750))

(wait-futures 1
              (while (flawed-loot smaug bilbo))
              (while (flawed-loot smaug gandalf)))
;= nil
(map (comp count :items deref) [bilbo gandalf])
;= (5 48)
(filter (:items @bilbo) (:items @gandalf))
;= (18 32 1)
```

Воспользовавшись теми же проверками, что представлены в примере 4.3, можно убедиться, что `flawed-loot` имеет некоторые проблемы: Бильбо получил 5 предметов, а Гендальф 48 (причем, предметы с числовыми идентификаторами `18`, `32` и `1` получили оба персонажа). Такая ситуация не должна получаться, потому что Смог имел ровно 50 предметов.

В чем же ошибка? В трех случаях одно и то же значение из слота `:items` Смога было помещено в слоты `:items` обоих персонажей, Бильбо и Гендальфа. Этого не случалось в заведомо правильной реализации `loot`, потому что использование функции `alter` гарантирует идентичность значений, полученных в транзакции и записываемых в ссылки при подтверждении.

В данном конкретном случае есть возможность обеспечить корректную передачу предметов с помощью `commute` (потому что порядок, в каком предметы будут передаваться, не имеет значения). Для этого нужно удалять переданные элементы из источника с помощью `alter`:

Пример 4.5. Функция fixed-loot, использующая commute и alter

```
(defn fixed-loot
  [from to]
  (dosync
```

```
    (when-let [item (first (:items @from))]
      (commute to update-in [:items] conj item)
      (alter from update-in [:items] disj item))))

(def smaug (character "Smaug" :health 500 :strength 400 :items (set (range 50))))
(def bilbo (character "Bilbo" :health 100 :strength 100))
(def gandalf (character "Gandalf" :health 75 :mana 750))

(wait-futures 1
              (while (fixed-loot smaug bilbo))
              (while (fixed-loot smaug gandalf)))
;= nil
(map (comp count :items deref) [bilbo gandalf])
;= (24 26)
(filter (:items @bilbo) (:items @gandalf))
;= ()
```

С другой стороны, `commute` идеально подходит для выполнения других операций в нашей игре. Например, функции `attack` и `heal` просто увеличивают и уменьшают соответствующие атрибуты персонажей, поэтому такие изменения безопасно выполнять с помощью `commute`:

```
(defn attack
  [aggressor target]
  (dosync
    (let [damage (* (rand 0.1) (:strength @aggressor))]
      (commute target update-in [:health] #(max 0 (- % damage))))))

(defn heal
  [healer target]
  (dosync
    (let [aid (* (rand 0.1) (:mana @healer))]
      (when (pos? aid)
        (commute healer update-in [:mana] - (max 5 (/ aid 5)))
        (commute target update-in [:health] + aid)))))
```

Создав пару дополнительных функций, мы сможем имитировать действия игрока в игре:

Пример 4.6. Функции имитации действия пользователя

```
(def alive? (comp pos? :health))

(defn play
```

```
  [character action other]
  (while (and (alive? @character)
              (alive? @other)
              (action character other))
    (Thread/sleep (rand-int 50))))          ❶
```

❶ Гарантирует, что никто не сможет выполнить более 20 действий в секунду!

Теперь можно провести поединок:

```
(wait-futures 1
              (play bilbo attack smaug)
              (play smaug attack bilbo))
;= nil
(map (comp :health deref) [smaug bilbo])    ❶
;= (488.80755445030337 -12.0394908759935)
```

❶ В одиночку Бильбо не сможет победить Смога.

...или «эпическую» битву:

Пример 4.7. Битва с участием трех персонажей

```
(dosync                                                    ❶
  (alter smaug assoc :health 500)
  (alter bilbo assoc :health 100))

(wait-futures 1
                (play bilbo attack smaug)
                (play smaug attack bilbo)
                (play gandalf heal bilbo))

;= nil
(map (comp #(select-keys % [:name :health :mana]) deref)
     [smaug bilbo gandalf])                                ❷
;= ({:health 0, :name "Smaug"}
;=  {:health 853.6622368542827, :name "Bilbo"}
;=  {:mana -2.575955687302212, :health 75, :name "Gandalf"})
```

❶ Вернуть персонажи в первоначальное физическое состояние.
❷ Бильбо сможет победить Смога, если Гендальф будет подпитывать его жизненной энергией в течение битвы.

Затирание состояния ссылки с помощью ref-set

Функция `ref-set` запишет значение, полученное в транзакции, в указанную ссылку:

```
(dosync (ref-set bilbo {:name "Bilbo"}))
;= {:name "Bilbo"}
```

Как и `alter`, функция `ref-set` перезапустит транзакцию, если состояние ссылки изменится к моменту подтверждения транзакции. Иначе говоря, семантика `ref-set` эквивалентна семантике вызова `alter` с функцией, возвращающей постоянное значение:

```
(dosync (alter bilbo (constantly {:name "Bilbo"})))
; {:name "Bilbo"}
```

Поскольку изменение выполняется без учета текущего состояния ссылки, очень легко попасть в ситуацию, когда произведенное изменение выглядит согласованным с точки зрения гарантий транзакционной памяти, но противоречит прикладной логике. Поэтому `ref-set` обычно используется только для повторной инициализации состояний ссылок начальными значениями.

Проверка локальной согласованности с помощью валидаторов

Если вы заметили, в конце примера 4.7 Бильбо имеет *очень* высокий показатель здоровья, значение `:health`. И действительно, мы никак не ограничиваем максимальное значение `:health` персонажей, которое растет в результате лечения и других восстановительных действий.

В играх подобного рода обычно не допускается, чтобы показатель здоровья персонажа превышал некоторый уровень. Однако, с технической точки зрения и с точки зрения управления – особенно в больших командах разработчиков или при большом объеме программного кода – может оказаться весьма затруднительно гарантировать невозможность бесконтрольного увеличения показателя здоровья в каждой функции. Предотвращение появления недопустимых значений можно сделать частью семантики таких функций, но предпочтительнее было бы иметь возможность защищать целостность нашей модели отдельно. Поддержку локальной согласован-

ности, как в данном случае, – буква «C»[1] в аббревиатуре «ACID» – в ходе конкурирующих изменений легко можно реализовать с помощью функций-валидаторов.

О функциях-валидаторах уже рассказывалось в разделе «Функции-валидаторы», выше. Особенности их использования со ссылками ничем не отличаются от особенностей использования с любыми другими ссылочными типами, но при взаимодействии с транзакционной памятью они становятся особенно удобны: если функция-валидатор обнаружит недопустимое состояние, сгенерированное ею исключение (как и любое другое исключение, сгенерированное в ходе транзакции) вызовет неудачное завершение транзакции.

Учитывая это, мы можем реорганизовать реализацию нашей игры. Во-первых, функцию `character` нужно изменить так, чтобы она:

1. Добавляла множество общих валидаторов ко всем персонажам.
2. Допускала передачу ей дополнительных валидаторов, чтобы иметь возможность накладывать ограничения в зависимости от класса персонажа, уровня сложности игры и других игровых параметров:

```
(defn- enforce-max-health                                              ❶
  [{:keys [name health]}]
  (fn [character-data]
    (or (<= (:health character-data) health)
      (throw (IllegalStateException.
              (str name " is already at max health!"))))))

(defn character
  [name & {:as opts}]
  (let [cdata (merge {:name name :items #{} :health 500}
                     opts)
        cdata (assoc cdata :max-health (:health cdata))                ❷
        validators (list* (enforce-max-health name (:health cdata))    ❸
                          (:validators cdata))]
    (ref (dissoc cdata :validators)
      :validator #(every? (fn [v] (v %)) validators))))                ❹
```

❶ `enforce-max-health` возвращает функцию, принимающую новое потенциальное состояние персонажа, которая возбуждает исключение, если новое значение атрибута `:health` превышает первоначальный показатель здоровья персонажа.

[1] Consistency – согласованность, непротиворечивость. – *Прим. перев.*

❷ Первоначальное значение атрибута `:health` персонажа копируется в его атрибут `:max-health`, который будет использоваться позднее.
❸ Помимо ограничения максимального значения показателя здоровья, легко можно наложить другие ограничения, характерные для данного персонажа, в виде дополнительного множества функций-валидаторов...
❹ ...которые легко можно задействовать для проверки содержащей их ссылки.

Теперь ни один персонаж не сможет получить значение показателя здоровья выше первоначального уровня:

```
(def bilbo (character "Bilbo" :health 100 :strength 100))
;= #'user/bilbo
(heal gandalf bilbo)
;= #<IllegalStateException java.lang.IllegalStateException: Bilbo is already
     at max health!>
```

Одно из ограничений валидаторов состоит в том, что они накладывают *локальные* ограничения. То есть, персонаж не может развиваться выше установленного уровня, так как новое состояние ссылки должно удовлетворять установленным ограничениям:

```
(dosync (alter bilbo assoc-in [:health] 95))
;= {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 95, :xp 0}
(heal gandalf bilbo)
;= #<IllegalStateException java.lang.IllegalStateException: Bilbo is already
     at max health!>
```

Этот фрагмент устанавливает показатель здоровья `:health` персонажа Бильбо чуть ниже его значения `:max-health`, поэтому в действительности должна иметься возможность увеличивать показатель его здоровья. Однако реализация `heal` не учитывает значение `:max-health`, и в валидаторе нет возможности «поправить» новое состояние Бильбо, чтобы оно удовлетворяло условиям, в данном случае – уменьшить значение `:health` или сумму текущего значения `:health` и значения, возвращаемого функцией `heal` Гендальфа, чтобы не превысить значение атрибута `:max-health`. Если бы валидаторы *имели* возможность вносить такие «поправки», было бы очень сложно сохранить согласованность значений ссылок, изменяемых в транзакциях. Валидаторы существуют исключительно для поддержания инвариантности модели.

Ниже представлена измененная версия функции heal, гарантирующая, что «прибавка» показателя здоровья персонажа не позволит превысить его максимальное значение:

```
(defn heal
  [healer target]
  (dosync
    (let [aid (min (* (rand 0.1) (:mana @healer))
                   (- (:max-health @target) (:health @target)))]
      (when (pos? aid)
        (commute healer update-in [:mana] - (max 5 (/ aid 5)))
        (alter target update-in [:health] + aid)))))
```

Теперь heal будет увеличивать показатель здоровья персонажа до уровня, *не выше* максимального значения, возвращая nil, если здоровье персонажа уже находится на максимальном уровне:

```
(dosync (alter bilbo assoc-in [:health] 95))
;= {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 95}
(heal gandalf bilbo)
;= {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 100}
(heal gandalf bilbo)
;= nil
```

Обратите внимание, что теперь величина target зависит от предыдущего состояния, поэтому здесь вместо commute используется alter. Строго говоря, это не является обязательным: возможно, вас вполне устроил бы валидатор, перехватывающий ошибочные значения, которые могут получаться, только если какая-нибудь другая конкурирующая транзакция также увеличивает показатель здоровья указанного персонажа. Все эти сложности указывают на недостаточную продуманность модели персонажей, спроектированных как единые блоки (в данном случае, как ассоциативные массивы) с информацией о состоянии: если конкурирующая транзакция изменит с помощью alter другую часть состояния, не имеющую отношения к текущей, это вызовет ненужный перезапуск текущей транзакции[1].

[1] Уточнение *детализации модели* относится к этапу оптимизации, на котором вы должны будете принять решение, основываясь на результатах экспериментов и некоторой доли предусмотрительности. Всегда начинайте с самого простого подхода – модели данных, построенные по принципу «все в одном», вполне пригодны в большинстве случаев – прибегать к более сложным решениям следует, только когда они действительно необходимы. Одно из возможных решений можно найти в статье http://clj-me.cgrand.net/2011/10/06/a-world-in-a-ref/.

Острые углы программной транзакционной памяти

Как отмечалось в начале этой главы, Clojure не предлагает универсального рецепта решения проблем конкурентного программирования. Иногда его реализация транзакционной памяти может казаться идеальной (и, в сравнении с типичными альтернативами, связанными с управлением блокировками вручную, так оно и есть), но даже транзакционная память имеет острые углы и грани, которых следует остерегаться.

Функции с побочными эффектами строго запрещены

В рамках транзакции должны выполняться только операции, которые безопасно можно повторить, что исключает возможность использования любых форм ввода/вывода. Например, если внутрь блока `dosync` включить операцию записи в файл или в базу данных, высока вероятность, что одни и те же данные будут записаны в файл или в базу данных многократно.

Clojure не способен выявлять попытки выполнения небезопасных операций внутри транзакций – он без тени сомнения повторит эти операции, что может привести к пагубным последствиям. По этой причине в Clojure имеется макрос `io!`, генерирующий ошибку при выполнении в рамках транзакции. То есть, если в программе имеется функция, которая может вызываться внутри транзакции, тогда завернув часть ее тела, дающую побочные эффекты, в форму `io!`, можно защититься от непреднамеренного выполнения небезопасного кода:

```
(defn unsafe
  []
  (io! (println "writing to database...")))

;= #'user/unsafe
(dosync (unsafe))
;= #<IllegalStateException java.lang.IllegalStateException: I/O in
     transaction>
```

Внимание. Как следствие, операции с атомами обычно следует рассматривать, как имеющие побочные эффекты, поскольку функция `swap!` и другие не поддерживают семантику транзакций. То есть, если транзакция,

содержащая вызов `swap!`, будет повторена трижды, она трижды вызовет `swap!` и целевой атом будет изменен трижды ...что едва ли часто бывает желательно, если только вы не собираетесь с помощью атома подсчитать количество повторений транзакции.

Обратите также внимание, что значения, хранимые ссылками, *должны* быть неизменяемыми[1]. Clojure не препятствует использованию изменяемых объектов в ссылках, но совокупность таких особенностей, как повторение транзакций и обычные недостатки, сопутствующие изменяемости, наверняка приведут к нежелательным эффектам:

```
(def x (ref (java.util.ArrayList.)))
;= #'user/x
(wait-futures 2 (dosync (dotimes [v 5]
                          (Thread/sleep (rand-int 50))        ❶
                          (alter x #(doto % (.add v))))))
;= nil
@x
;= #<ArrayList [0, 0, 1, 0, 2, 3, 4, 0, 1, 2, 3, 4]>        ❷
```

❶ Вызов `sleep` для приостановки на случайно выбранный интервал времени гарантирует пересечение двух транзакций – по крайней мере одна из них будет повторена, что приведет...

❷ ...к безнадежно ошибочным результатам.

Минимизируйте продолжительность выполнения транзакций

Как уже говорилось в обсуждении вокруг рис. 4.4, задача поддержки транзакционной памяти состоит в том, чтобы гарантировать последовательное выполнение операций со ссылками, предусматриваемых транзакцией, переупорядочивая эти операции при необходимости. Из этого следует, что чем короче будет каждая транзакция, тем меньше будут накладные расходы, связанные с их обслуживанием, и выше скорость и пропускная способность приложения.

[1] Или, по крайней мере, должны обеспечивать высокую производительность при их изменении. Например, в качестве состояния ссылки вполне можно использовать список Java, если строго придерживаться приема «копирования при записи» (copy-on-write) для создания измененных версий списков, но это не только дурной тон, но и практически никогда не требуется.

Что случится, если транзакции будут слишком протяженными, или в приложении будут присутствовать транзакции самой разной протяженности? В общем случае самая протяженная транзакция окажется отложенной (а также все операции в потоке, ожидающем завершения транзакции, которые при другом раскладе могли бы быть выполнены). Взгляните на рис. 4.5, где изображено множество транзакций, каждая из которых изменяет состояние ссылки `a`:

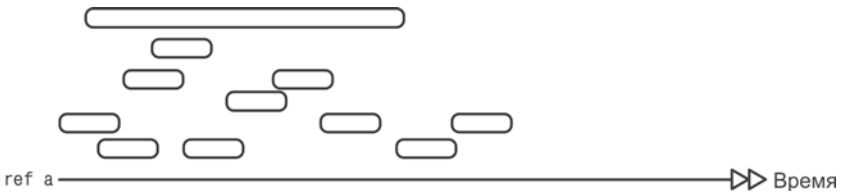

Рис. 4.5. Множество транзакций, изменяющих состояние ссылки `a`

Если допустить, что каждая из них изменяет `a` с помощью функции `alter`, тогда все эти транзакции будут повторяться, пока не смогут быть выполнены последовательно. Самая протяженная транзакция будет повторяться несколько раз, что эквивалентно задержке до момента, пока не откроется достаточно протяженный интервал времени в шкале времени жизни ссылки:

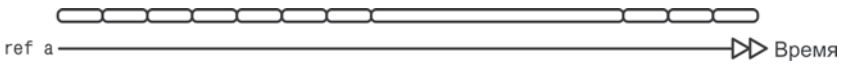

Рис. 4.6. Фактический порядок применения транзакций к ссылке `a`

Примечание. Не забывайте, что функция `commute` (обсуждавшаяся в разделе «Уменьшение конфликтов в транзакциях с помощью commute», выше) позволяет избавиться от конфликтов и повторений. Благодаря ей, при наличии возможности безопасно использовать ее с функциями, изменяющими состояние, можно обойти все потенциальные опасности, связанные с продолжительными транзакциями.

Продолжительные вычисления увеличивают протяженность транзакции и вероятность повторения из-за конкуренции за другие ссылки. Например, протяженная транзакция, изображенная на рис. 4.5, может выполнять некоторые сложные вычисления и не-

однократно перезапускаться из-за конкуренции за другие ссылки. Поэтому, необходимо стараться уменьшать протяженность транзакций, насколько это возможно, и уменьшать количество вовлеченных в работу ссылок.

Живая блокировка (live lock). Кого-то может беспокоить вопрос: «Не случится ли так, что в моменты высокой нагрузки, протяженная транзакция не получит шанс подтвердить свои изменения из-за конкуренции за ссылку?». Это – ситуация в механизме транзакционной памяти, называемая *живой блокировкой* (live lock) и эквивалентная ситуации взаимоблокировки (deadlock), когда поток выполнения, управляющий транзакцией, оказывается заблокированным на неопределенный интервал времени из-за невозможности подтвердить свою транзакцию. Без соответствующих обходных путей, даже управляя блокировками вручную, мы просто сталкивались бы с взаимоблокировками!

К счастью, реализация транзакционной памяти имеет пару обходных путей. Первый называется *проталкивание* (barging), когда в некоторых случаях более старой транзакции позволяется завершить работу, а более новые транзакции принудительно повторяются в это время. Если прием проталкивания не даст выполнить самую старую транзакцию за приемлемый период времени, механизм транзакционной памяти просто заставит ее потерпеть неудачу:

```
(def x (ref 0))
;= #'user/x
(dosync
  @(future (dosync (ref-set x 0)))
  (ref-set x 1))
;= #<RuntimeException java.lang.RuntimeException:
;= Transaction failed after reaching retry limit>
@x
;= 0
```

Транзакция в примере выше, выполняющаяся в потоке, управляющем оболочкой REPL, всегда запускает новый объект `future`, который сам выполняет транзакцию, изменяющую состояние совместно используемой ссылки. Операция разыменования `future` гарантирует, что транзакция, выполняемая в потоке, управляющем оболочкой REPL, будет ждать завершения другой транзакции, постоянно перезапускаясь и тем самым запуская новый объект `future`, и так далее.

Механизм транзакционной памяти в Clojure разрешает транзакции перезапуститься не более установленного количества раз, а затем возбуждает исключение. Появление ошибки вместе с трассировкой стека, который можно исследовать, несомненно лучше, чем взаимоблокировка (или живая блокировка), когда единственное, что остается, – это принудительно прервать выполнение процесса и не получить никакой информации о проблеме.

Читающие транзакции могут повторяться

При применении к ссылочным типам, функция `deref` никогда не блокируется. Однако *внутри транзакции* операция разыменования ссылки может стать причиной ее повторения!

Причина в следующем: если после запуска текущей транзакции ссылка получит новое значение, подтвержденное другой транзакцией, *прежнее значение ссылки, имевшее место на момент начала текущей транзакции,* невозможно будет получить[1]. К счастью, механизм транзакционной памяти предусматривает решение этой проблемы и поддерживает ограниченную историю состояний ссылок, вовлеченных в транзакцию, размер которой увеличивается с каждым повторением. Это увеличивает вероятность, что в некоторый момент транзакцию не придется больше повторять, потому что несмотря на параллельные изменения ссылки, желаемое значение будет присутствовать в истории.

Узнать или изменить длину истории можно с помощью функций `ref-history-count`, `ref-max-history` и `ref-min-history`. Минимальный и максимальный размеры истории можно также задать в момент создания ссылки, с помощью именованных аргументов `:min-history` и `:max-history`:

```
(ref-max-history (ref "abc" :min-history 3 :max-history 30))
;= 30
```

Это позволяет подготовить ссылку к ожидаемым нагрузкам.

Повторения из-за `deref` обычно происходят в контексте транзакций, выполняющих только операции чтения, которые пытаются сделать моментальный снимок большого количества очень активных

[1] См. раздел «Искажение при записи», ниже, где приводятся дополнительные подробности, касающиеся значений, возвращаемых функцией `deref` внутри транзакции.

ссылок. Продемонстрировать эту ситуацию можно на примере единственной ссылки и длительной транзакцией, выполняющей чтение:

```
(def a (ref 0))
(future (dotimes [_ 500] (dosync (Thread/sleep 200) (alter a inc))))
;= #<core$future_call$reify__5684@10957096: :pending>
@(future (dosync (Thread/sleep 1000) @a))
;= 28                                    ❶
(ref-history-count a)
;= 5
```

❶ Значение 28 говорит о том, что читающая транзакция смогла завершиться еще до того, как были запущены все пишущие транзакции.

Итак, история состояний ссылки `a` продолжала расти до тех пор, пока не оказалось возможным выполнить медленную читающую транзакцию. А что произойдет, если операции записи будут выполняться чаще?

```
(def a (ref 0))
(future (dotimes [_ 500] (dosync (Thread/sleep 20) (alter a inc))))
;= #<core$future_call$reify__5684@10957096: :pending>
@(future (dosync (Thread/sleep 1000) @a))
;= 500
(ref-history-count a)
;= 10
```

На этот раз размер истории увеличился больше, и читающая транзакция выполнилась только после завершения всех пишущих транзакций. Это означает, что пишущие транзакции заблокировали читающую транзакцию. Если увеличить максимальный размер истории, проблема должна решиться:

```
(def a (ref 0 :max-history 100))
(future (dotimes [_ 500] (dosync (Thread/sleep 20) (alter a inc))))
;= #<core$future_call$reify__5684@10957096: :pending>
@(future (dosync (Thread/sleep 1000) @a))
;= 500
(ref-history-count a)
;= 10
```

Это решение не сработало, потому что к моменту, когда история достигла достаточного размера, все пишущие транзакции уже завер-

шились. Поэтому здесь важно выбрать оптимальную минимальную длину истории:

```
(def a (ref 0 :min-history 50 :max-history 100))    ❶
(future (dotimes [_ 500] (dosync (Thread/sleep 20) (alter a inc))))
@(future (dosync (Thread/sleep 1000) @a))
;= 33
```

❶ Значение 50 было выбрано потому, что читающая транзакция в 50 раз медленнее любой пишущей транзакции.

На этот раз читающая транзакция завершилась быстрее и дело обошлось без повторов!

Искажение при записи

Механизм транзакционной памяти в Clojure предназначен для обеспечения согласованности состояний ссылок в транзакциях, но до сих пор мы рассматривали только ситуации, когда ссылки изменяются в транзакциях. Если ссылка не изменяется в транзакции, но правильность изменения других ссылок зависит от состояния ссылки, не изменяющегося в транзакции, механизм транзакционной памяти не сможет узнать об изменении данной ссылки, выполненном с помощью вызовов `alter`, `commute` и `set-ref`. Если в середине транзакции изменится состояние прочитанной ссылки, от которого зависит правильность изменения состояний других ссылок, дело может закончиться записью в другие ссылки значений, противоречащих значению читаемой ссылки. Такую ситуацию называют *искажением при записи* (write skew).

Такие случаи встречаются редко – обычно изменяются все ссылки, вовлеченные в транзакцию. Однако, в подобных редких случаях для предотвращения искажения при записи можно использовать функцию `ensure`: она позволяет разыменовывать ссылку так, что если обнаружится конфликт с любыми изменениями, произведенными другими транзакциями, эти транзакции будут перезапущены.

Примером такой ситуации в контексте игры может служить степень освещенности. Можно смело сказать, что поединок под полуденным солнцем имеет больше шансов завершиться победой, чем ночью, поэтому есть определенный смысл изменить функцию `attack` так, чтобы она учитывала освещенность поля боя:

```
(def daylight (ref 1))

(defn attack
  [aggressor target]
  (dosync
    (let [damage (* (rand 0.1) (:strength @aggressor) @daylight)]
      (commute target update-in [:health] #(max 0 (- % damage))))))
```

Однако, если состояние ссылки `daylight` изменится между моментом, когда она будет прочитана внутри транзакции, и моментом, когда транзакция подтвердит изменения, эти изменения могут оказаться несогласованными. Например, в игре может выполняться отдельный процесс, изменяющий освещенность в зависимости от времени суток (например, `(dosync (ref-set daylight 0.3))`). Если значение освещенности будет изменено во время выполнения функции `attack`, а в вычислениях будет использоваться прежнее значение `daylight`, атакующий незаконно получит лишнее преимущество.

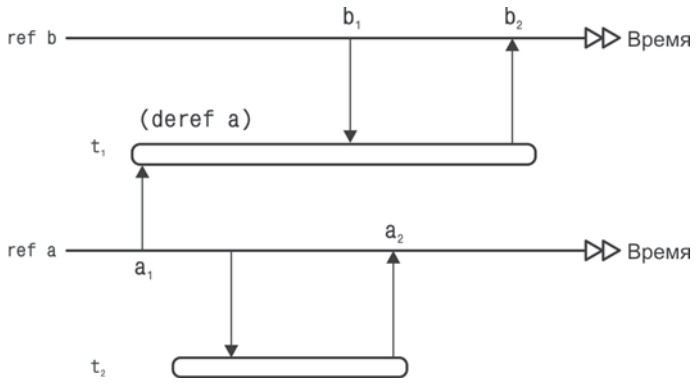

Рис. 4.7. Искажение при записи, где состояние b_2 зависит от состояния `a`, прочитанного ранее

Формально, если состояние b_2, которое транзакция t_1 записывает в ссылку `b`, зависит от состояния a_1 ссылки `a`, и t_1 не изменяет ссылку `a`, и другая транзакция t_2 изменяет ссылку `a`, записывая в нее состояние a_2 до того, как t_1 подтвердит изменения, тогда значения ссылок окажутся несогласованными: состояние b_2 будет соответствовать прежнему состоянию a_1 ссылки `a`, а не текущему a_2. Эта и есть искажение при записи.

Простая замена операции разыменования `@daylight` на `(ensure daylight)` в функции `attack` позволит избежать этого, гарантируя неизменность уровня освещенности, пока не завершится читающая транзакция.

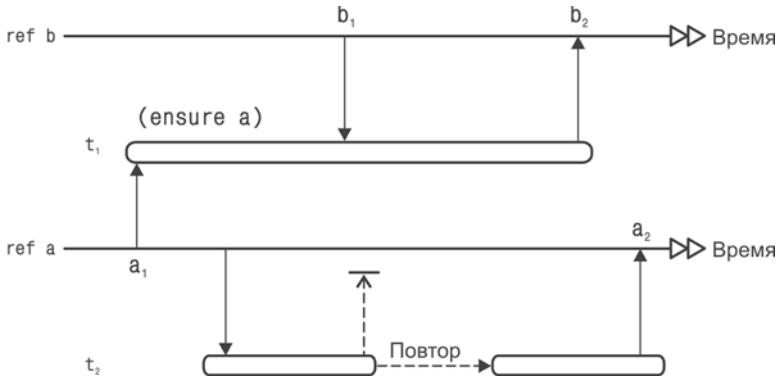

Рис. 4.8. Избавление от искажения при записи с помощью `ensure`

Когда транзакция t_1 прочитает ссылку `a` с помощью `ensure` вместо `deref`, любые попытки изменить состояние этой ссылки в другой транзакции t_2 до завершения транзакции t_1 будут повторяться, пока t_1 не завершится. Это дает возможность избежать искажения при записи: изменения в ссылке `b` всегда будут соответствовать последнему состоянию ссылки `a`, даже при том, что t_1 не изменяет состояние ссылки `a`.

Примечание. В терминах исключения вероятности искажения при записи, вызов `(ensure a)` семантически эквивалентен вызову `(alter a identity)` или `(ref-set a @a)` – фиктивные операции записи – что гарантирует сохранность прочитанного значения до момента завершения транзакции. Однако в отличие от фиктивных операций записи, `ensure` минимизирует общее число повторений транзакций, вовлекающих в работу ссылки, используемые только для чтения.

Переменные

Вы уже много раз сталкивались с переменными. *Переменные* отличаются от других ссылочных типов тем, что они предназначены не для управления изменением их состояния во времени, а скорее для

представления идентичности, глобальной для пространства имен, которой при необходимости можно привязывать разные значения *в разных потоках выполнения*. Подробнее об этом будет рассказываться, начиная с раздела «Динамическая область видимости», ниже, но сначала необходимо разобраться с некоторыми основными особенностями переменных, потому что они используются в языке Clojure повсюду, независимо от того, идет ли речь о конкуренции или нет.

Определение значения символа в Clojure обычно сводится к поиску переменной с этим именем в текущем пространстве имен и разыменованию переменной, с целью получить ее значение. Однако также возможно получить ссылку на переменную и вручную разыменовать ее:

```
map
;= #<core$map clojure.core$map@501d5ebc>
#'map                                          ❶
;= #'clojure.core/map
@#'map
;= #<core$map clojure.core$map@501d5ebc>
```

❶ В разделе «Ссылки на переменные: var», в главе 1, уже говорилось, что форма записи `#'map` – это всего лишь синтаксический сахар, эквивалентный форме записи `(var map)`.

Определение переменных

Переменные являются одним из основных строительных блоков в языке Clojure. Как упоминалось в разделе «Определение переменных: def», в главе 1, все функции и значения верхнего уровня сохраняются в переменных, которые определяются в текущем пространстве имен с использованием специальной формы `def` или одной из ее производных.

Помимо простого создания переменной в пространстве имен с указанным именем, `def` копирует также метаданные[1], имеющиеся в символе, представляющем имя для новой (или обновляемой) переменной в саму переменную. Некоторые метаданные, которые будут перечислены далее и имеющиеся в этом символе, могут влиять на поведение и семантику переменной.

[1] Пример метаданных в языке Clojure можно найти в разделе «Метаданные», в главе 3.

Приватные переменные

Приватные переменные (private vars) являются основным способом обозначить границы разделов библиотеки или API, которые зависят от особенностей реализации или не предназначены для непосредственного использования внешними пользователями. Приватные переменные:

1. Доступны из других пространств имен, только при использовании полностью квалифицированных имен.
2. Их значения можно получить, только выполнив разыменование вручную.

Переменная становится приватной, если символ с этим именем имеет слот `:private` в ассоциативном массиве с метаданными со значением `true`. Ниже приводится пример определения приватной переменной, хранящей некоторое постоянное значение, которое может понадобиться в нашем коде:

```
(def ^:private everything 42)
```

Как уже говорилось в разделе «Метаданные», в главе 3, такая форма записи эквивалентна следующей нотации:

```
(def ^{:private true} everything 42)
```

Нетрудно убедиться, что для доступа к приватным переменным из других пространств имен необходимо приложить дополнительные усилия:

```
(def ^:private everything 42)
;= #'user/everything
(ns other-namespace)
;= nil
(refer 'user)
;= nil
everything
;= #<CompilerException java.lang.RuntimeException:
;= Unable to resolve symbol: everything in this context, compiling:(NO_SOURCE_
PATH:0)>
@#'user/everything
;= 42
```

Объявить приватную функцию можно с помощью формы `defn-`, которая полностью идентична знакомой нам форме `defn`, за исключением того, что она автоматически добавляет `^:private` в метаданные.

Строки документации

Clojure позволяет снабжать переменные верхнего уровня документацией с помощью *строк документации* (docstrings), которые представляют собой обычные строковые литералы, следующие непосредственно за символом с именем переменной:

```
(def a
  "A sample value."
  5)
;= #'user/a
(defn b
  "A simple calculation using `a`."
  [c]
  (+ a c))
;= #'user/b
(doc a)
; -------------------------
; user/a
;   A sample value.
(doc b)
; -------------------------
; user/b
; ([c])
;   A simple calculation using `a`.
```

Как видите, строки документации – это всего лишь одна из разновидностей метаданных; однако за кулисами форма `def` выполняет некоторые дополнительные операции по добавлению строк документации в метаданные переменных:

```
(meta #'a)
;= {:ns #<Namespace user>, :name a, :doc "A sample value.",
;=  :line 1, :file "NO_SOURCE_PATH"}
```

Это означает, что добавлять документацию в переменные можно также явно определив значение слота `:doc`, как во время объявления переменной, так и после, изменив метаданные этой переменной:

```
(def ^{:doc "A sample value."} a 5)
;= #'user/a
(doc a)
; -------------------------
; user/a
```

```
;   A sample value.
(alter-meta! #'a assoc :doc "A dummy value.")
;= {:ns #<Namespace user>, :name a, :doc "A dummy value.",
;=  :line 1, :file "NO_SOURCE_PATH"}
(doc a)
; -------------------------
; user/a
;   A dummy value.
```

Такой прием редко используется на практике, но он может пригодиться при создании макросов определения переменных.

Константы

Довольно часто в программе требуется определять *константы*, для чего обычно используются формы `def` верхнего уровня. Чтобы компилятор интерпретировал имя переменной как константу, в метаданные можно добавить слот `^:const`:

```
(def ^:const everything 42)
```

Помимо того, что слот `^:const` является дополнительным документирующим признаком, он также оказывает влияние на поведение переменной: ссылки на такие константы разрешаются не на этапе выполнения (как это делается обычно), а во время компиляции – значение константы подставляется компилятором непосредственно в код. Это не только дает некоторый прирост производительности, но, что более важно, гарантирует *неизменность* константы, даже если позднее кто-то попытается изменить значение такой переменной.

Следующий пример демонстрирует непредусмотренное использование переменной:

```
(def max-value 255)
;= #'user/max-value
(defn valid-value?
  [v]
  (<= v max-value))
;= #'user/valid-value?
(valid-value? 218)
;= true
(valid-value? 299)
;= false
(def max-value 500)     ❶
```

```
;= #'user/max-value
(valid-value? 299)
;= true
```

❶ Выполняется переопределение переменной `max-value`, после чего `valid-value?` приобретает иную семантику из-за того, что полагается на нашу «константу».

Предотвратить этот беспорядок можно с помощью `^:const`:

```
(def ^:const max-value 255)
;= #'user/max-value
(defn valid-value?
  [v]
  (<= v max-value))
;= #'user/valid-value?
(def max-value 500)
;= #'user/max-value
(valid-value? 299)
;= false
```

Поскольку теперь переменная `max-value` объявлена с указанием слота `^:const`, ее значение будет сохранено в функции `valid-value?` на этапе компиляции. Любые последующие изменения значения `max-value` не будут оказывать влияния на семантику `valid-value?`, пока эта функция не будет повторно переопределена.

Динамическая область видимости

Большинство имен в языке Clojure имеют лексическую область видимости: то есть имена получают значения в зависимости от определений форм, ограничивающих их использование, и пространств имен, где они вычисляются. Например:

```
(let [a 1
      b 2]
  (println (+ a b))          ❶
  (let [b 3
        + -]
    (println (+ a b))))      ❷
;= 3
;= -2
```

❶ a и b – это имена локальных переменных, созданных формой let; + и println – это имена переменных, содержащих функции, объявленные в пространстве имен clojure.core, доступном внутри текущего пространства имен.

❷ Локальная переменная b связывается с другим значением, как и переменная +; так как эти определения лексически являются более локальными, чем внешняя локальная переменная b и оригинальная переменная с именем +, они маскируют первоначальные значения, при обращении к ним в данном контексте.

Исключением из этого правила является *динамическая область видимости* (dynamic scope) – особенность, поддерживаемая переменными. Переменные имеют *корневую привязку* (root binding); то есть значение, которое привязывается к переменной при ее определении с помощью формы def или некоторых ее производных и которое возвращается при обращении к переменной. Однако, если переменная определяется как *динамическая* (с использованием слота ^:dynamic в метаданных), корневая привязка может переопределяться и маскироваться в потоках выполнения с использованием формы binding[1].

```
(def ^:dynamic *max-value* 255)
;= #'user/*max-value*
(defn valid-value?
  [v]
  (<= v *max-value*))
;= #'user/valid-value?
(binding [*max-value* 500]
  (valid-value? 299))
;= true
```

Примечание. Динамические переменные, предназначенные для повторной привязки значений с помощью binding должны окружаться звездочками, например: *this*, известными также как «наушники» (earmuffs). Это – всего лишь соглашение об именовании, но оно позволяет предупредить того, кто будет читать код, что переменная может иметь динамическую область видимости.

Ниже демонстрируется возможность изменить значение переменной *max-value* с помощью binding за пределами лексической области

[1] Попытка применить binding к переменной без слота :dynamic в метаданных приведет к исключению.

видимости внутри `valid-value?`. Однако новое значение будет доступно только в данном потоке выполнения – в других потоках значение `*max-value*` останется прежним[1]:

```
(binding [*max-value* 500]
  (println (valid-value? 299))
  (doto (Thread. #(println "in other thread:" (valid-value? 299)))
    .start
    .join))
;= true
;= in other thread: false
```

Динамические области видимости широко используются в библиотеках и в самом Clojure[2] с целью обеспечить возможность изменения конфигурации по умолчанию без явного использования многопоточного контекста в вызове каждой функции. Весьма интересные примеры можно также найти в главах 14 и 15, где динамическая область видимости используется для передачи параметров настройки соединения с базой данных в библиотеку.

Визуализация динамической области видимости. Для иллюстрации рассмотрим переменную: она имеет корневое значение (root value) и в каждом потоке выполнения она *может* любое количество раз связываться с локальными для потока значениями, маскирующими друг друга в результате образования вложенных динамических областей видимости при применении формы `binding`.

Программе доступны только головы этих стопок (на которые указывают стрелки, на рис. 4.9). Как только будет выполнена привязка нового значения, предыдущая привязка маскируется до выхода из динамической области видимости, образованной формой `bind-`

---

[1] Приношу свои извинения за использование механизмов взаимодействий с Java. Чтобы продемонстрировать эту особенность динамических переменных, необходимо было воспользоваться «родными» (native) потоками выполнения. Описание происходящего в данном примере можно найти в разделе «Механизмы параллельного выполнения в Java», ниже, и в главе 9.

[2] В качестве примеров можно привести `*warn-on-reflection*`, как описывается в разделе «Указание типов для производительности», в главе 9, и в разделе «Ошибки и предупреждения, вызванные несоответствием типов», в главе 11. Переменные `*out*`, `*in*` и `*err*`, а также косвенное использование `binding` в таких функциях, как `with-precision`, обсуждается в разделе «Режимы масштабирования и округления в операциях с вещественными числами произвольной точности», в главе 11.

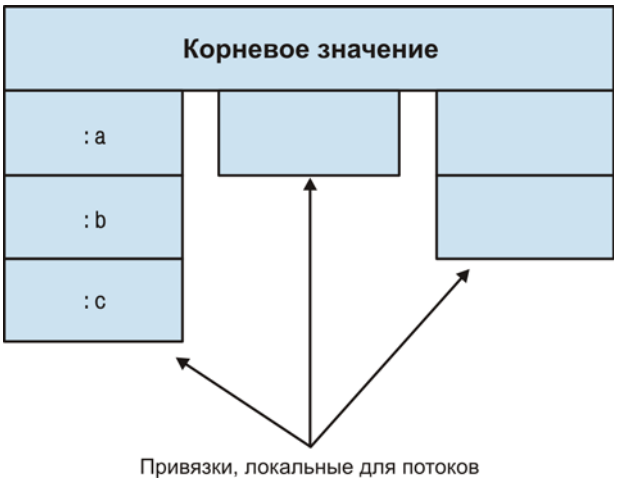

Рис. 4.9. Переменная, хранящая единственное корневое значение и множество локальных для потоков выполнения привязок, маскирующих друг друга

ing. Поэтому в самой внутренней динамической области видимости в данном примере, обращение к `*var*` (и соответственно `(get-*var*)`) никогда не даст значение `:root`, `:a` или `:b`:

```
(def ^:dynamic *var* :root)
;= #'user/*var*
(defn get-*var* [] *var*)
;= #'user/get-*var*
(binding [*var* :a]
  (binding [*var* :b]
    (binding [*var* :c]
      (get-*var*))))
;= :c
```

Для каждой новой динамической области видимости создается новый «кадр» стека:

```
(binding [*var* :a]
  (binding [*var* :b]
    (binding [*var* :c]
      (binding [*var* :d]
        (get-*var*)))))
;= :d
```

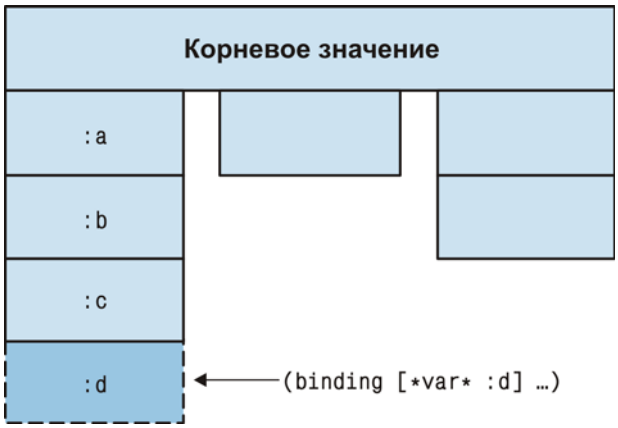

Рис. 4.10. Результат привязки к переменной нового локального значения с помощью `binding`

Мы уже видели, как можно использовать динамическую область видимости для дистанционного управления поведением функций, фактически позволяя вызывающей программе неявно передавать дополнительные аргументы, образуя новые уровни в дереве вызовов. Заключительным фрагментом мозаики является возможность работы динамических областей видимости в обратном порядке, что позволяет функциям предоставлять возвращать разные значения на разных уровнях в дереве вызовов.

Например, в языке Clojure имеется ряд чрезвычайно удобных функций ввода/вывода, извлекающих содержимое URL (например, `slurp` и другие функции в пространстве имен `clojure.java.io`), однако эти методы не дают простой возможности получить соответствующий HTTP-код ответа (что иногда бывает необходимо, особенно при использовании различных HTTP API). Одно из возможных решений состоит в том, чтобы *всегда* возвращать код ответа в векторе `[response-code url-content]` вместе с содержимым URL:

```
(defn http-get
  [url-string]
  (let [conn (-> url-string java.net.URL. .openConnection)
        response-code (.getResponseCode conn)]
    (if (== 404 response-code)
      [response-code]
```

```
      [response-code (-> conn .getInputStream slurp)])))

(http-get "http://google.com/bad-url")
;= [404]
(http-get "http://google.com/")
;= [200 "<!doctype html><html><head>..."]
```

В таком решении нет ничего страшного, но как пользователи `http-get`, мы будем вынуждены иметь дело с кодом ответа при каждом вызове, в любом контексте, даже когда этот код нам не нужен.

Альтернативное решение заключается в использовании динамической области видимости, которую можно создавать для `http-get`, *только когда желательно получить HTTP-код ответа*:

```
(def ^:dynamic *response-code* nil)                                          ❶

(defn http-get
  [url-string]
  (let [conn (-> url-string java.net.URL. .openConnection)
        response-code (.getResponseCode conn)]
    (when (thread-bound? #'*response-code*)                                  ❷
      (set! *response-code* response-code))                                  ❸
    (when (not= 404 response-code) (-> conn .getInputStream slurp))))        ❹

(http-get "http://google.com")
;= "<!doctype html><html><head>..."
*response-code*
;= nil
(binding [*response-code* nil]
  (let [content (http-get "http://google.com/bad-url")]
    (println "Response code was:" *response-code*)
    ; ... выполнить операции с `содержимым` если это не nil ...
    ))
;= Response code was: 404
;= nil
```

❶ Определяется новая переменная `*response-code*`; пользователи функции `http-get` могут использовать ее для получения кода ответа.

❷ С помощью функция `thread-bound?` проверяется, была ли выполнена локальная привязка значения к переменной `*response-code*` вызывающим кодом. Если нет, то она никак не используется.

❸ Функция `set!` используется для *изменения* значения текущей привязки `*responsecode*`, чтобы вызывающий код мог получить доступ к требуемому значению.

❹ Теперь, когда для передачи дополнительной информации вызывающему коду `http-get` использует необязательную динамическую область видимости, окружающую `*response-code*`, она может просто вернуть строку с содержимым URL, не создавая для этого вектор `[response-code urlcontent]` (если код возврата не равен 404).

Проиллюстрируем происходящее (рис. 4.11).

Рис. 4.11. Передача дополнительного значения из функции

Функция `set!` воздействует непосредственно на привязку переменной, замещая текущее локальное для потока значение, поэтому вызывающий код, создавший динамическую область видимости с помощью `binding` – непосредственно или 50 кадрами стека выше – сможет получить доступ к новому значению без необходимости продираться через значения, возвращаемые всеми промежуточными вызовами. Этот прием можно использовать с любым количеством переменных, с любым количеством привязок и с любым количеством значений, установленных функцией `set!`, включая и функции. Такая гибкость упрощает добавление расширений API, от организации возврата дополнительных значений, как было показано выше, до более сложных и мощных нелокальных механизмов возврата.

Динамические области видимости простираются через различные формы параллельного выполнения в Clojure. То обстоятельство, что динамические области видимости являются локальными для потоков выполнения, имеет свои преимущества – это позволяет изолировать друг от друга разные контексты выполнения – но это может также стать причиной ненужных сложностей при использовании средств языка Clojure, передающих вычисления из одного потока в другой. К счастью, динамические привязки переменных могут передаваться между потоками выполнения – этот прием называется *передачей привязки* (binding conveyance) – с помощью агентов (по-

средством send и send-off), объектов future, а также с применением функции pmap и ее вариантов:

```
(binding [*max-value* 500]
  (println (valid-value? 299))
  @(future (valid-value? 299)))
; true
;= true
```

Даже при том, что valid-value? вызывается в отдельном потоке выполнения, отличном от того, где с помощью binding определена динамическая область видимости, тем не менее, future передает эту область видимости из одного потока в другой на время его действия.

Обратите внимание, что хотя pmap поддерживает передачу привязки, она не поддерживается «ленивыми» последовательностями:

```
(binding [*max-value* 500]
  (map valid-value? [299]))
;= (false)
```

Чтобы решить эту проблему, необходимую динамическую область видимости необходимо создавать на каждом шаге реализации значений в «ленивой» последовательности:

```
(map #(binding [*max-value* 500]
         (valid-value? %))
     [299])
;= (true)
```

Переменные в языке Clojure не являются переменными в классическом понимании

Переменные (vars) в языке Clojure не следует путать с переменными (variables) в других языках программирования. Пришедшим из других языков программирования, таких как Ruby, где программный код обычно выглядит, как показано ниже:

```
def foo
  x = 123
  y = 456
  x = x + y
end
```

Невероятно трудно удержаться от соблазна попытаться писать на языке Clojure такой код:

```
(defn never-do-this []
  (def x 123)
  (def y 456)
  (def x (+ x y)
  x))
```

Такой подход не приветствуется в Clojure. Но что плохого в этом коде?

```
(def x 80)
;= #'user/x
(defn never-do-this []
  (def x 123)
  (def y 456)
  (def x (+ x y))
  x)
;= #'user/never-do-this
(never-do-this)
;= 579
x                          ❶
;= 579
```

❶ «Но постойте, вначале я присвоил переменной x значение 80!»

Форма def всегда определяет переменные *верхнего уровня*, это – *не* операция присваивания, воздействующая на некоторую локальную область видимости. Переменные x и y в этом примере являются глобально доступными из любого места в пространстве имен, и эти операции затронут любые другие переменные x и y, уже имеющиеся в этом пространстве имен.

Кроме переменных с динамической областью видимости, все остальные переменные в основном предназначены для хранения констант, от момента их объявления и до завершения приложения, REPL и так далее. Используйте другие ссылочные типы для хранения идентичностей, предоставляющие соответствующую семантику изменения на месте, если именно это вам требуется. Определяйте переменные для их хранения и используйте соответствующие функции (swap!, alter, send, sendoff и другие) для изменения состояний этих идентичностей.

Изменение корневой привязки переменной. Несмотря на различные предупреждения против использования переменных в качестве классических переменных, знакомых нам по другим языкам программирования, иногда бывает удобно иметь возможность изменять (с большой осторожностью) их корневые привязки. Чтобы изменить корневую привязку переменной, отталкиваясь от ее текущего значения, можно воспользоваться функцией `alter-var-root`:

```
(def x 0)
;= #'user/x
(alter-var-root #'x inc)
;= 1
```

Когда рассматриваемая переменная содержит функцию, данный прием предоставляет надмножество функциональности, которую можно найти в большинстве аспектно-ориентированных фреймворков. Конкретные примеры в этом направлении можно найти в разделе «Аспектно-ориентированное программирование», в главе 12, и в разделе «Сборка гибридных проектов», в главе 8.

Имеется также возможность временно изменять корневые привязки сразу нескольких переменных с помощью функции `with-redefs`, которая восстановит прежние корневые привязки переменных после выхода из ее области видимости. Это может пригодиться во время тестирования для определения фиктивных функций или значений, в зависимости от окружающего контекста. Примеры использования этой функции приводятся в разделе «Создание фиктивных значений», в главе 13.

Опережающие объявления

Существует возможность не определять значение переменной. В этом случае переменная считается «несвязанной» и при ее разыменовании будет возвращаться замещающий (placeholder) объект:

```
(def j)
;= #'user/j
j
;= #<Unbound Unbound: #'user/j>
```

Это удобно, когда требуется сослаться на переменную, пока не имеющую значения. Например, при реализации некоторых типов

алгоритмов, использующих прием чередующейся рекурсии, или когда просто желательно поместить реализацию функции ниже того места, где она используется, из соображения стиля оформления или с целью привлечь внимание к основным или общедоступным членам API. Компилятор Clojure компилирует и вычисляет формы в порядке их следования в исходных файлах, поэтому прежде чем сослаться на переменную, ее необходимо определить. Если допустить, что значения таких переменных будут необходимы только во время выполнения (например, если они играют роль контейнеров для функций), тогда фактические значения таким переменным можно присвоить позднее. Этот прием называется *опережающим объявлением* (forward declaration).

В подобных ситуациях более идиоматичным выглядит применение макроса `declare`. Его использование вместо формы `def` позволяет более явно выразить намерение объявить несвязанную переменную (уменьшая вероятность ситуаций, когда вы просто забыли указать значение), а кроме того дает возможность объявить множество несвязанных переменных в одном выражении:

```
(declare complex-helper-fn other-helper-fn)                          ❶

(defn public-api-function
  [arg1 arg2]
  ...
  (other-helper-fn arg1 arg2 (complex-helper-fn arg1 arg2))          ❷

(defn- complex-helper-fn                                             ❸
  [arg1 arg2]
  ...)

(defn- other-helper-fn
  [arg1 arg2 arg3]
  ...)
```

❶ Предварительное объявление переменных, представляющих вспомогательные функции.

❷ Теперь можно разместить основные/общедоступные члены API ближе к началу файла с исходным кодом и свободно ссылаться на вспомогательные функции.

❸ Определение вспомогательных функций далее в файле с исходным кодом.

Агенты

Агенты (agents) – это нескоординированный асинхронный ссылочный тип. Это означает, что изменения состояний одних агентов не зависят от изменений состояний других агентов, и что все изменения выполняются за пределами потоков выполнения, инициировавших их. Кроме того, агенты имеют две характеристики, отличающие их от атомов и ссылок:

1. В соединении с агентами можно безопасно использовать функции ввода/вывода и другие функции с побочными эффектами.
2. Агенты поддерживают транзакционную память, благодаря чему их смело можно использовать в контексте повторяющихся транзакций.

Изменять состояния агентов можно с помощью двух функций, `send` и `send-off`. Они следуют тому же шаблону, что и другие функции, изменяющие состояния ссылок – принимают другую функцию, возвращающую новое состояние агента и принимающую аргумент с его текущим состоянием, а также дополнительные аргументы.

Каждая совокупность функции с множеством необязательных аргументов, передаваемых функции `send` или `send-off`, называется *заданием* агента (agent action), а каждый агент поддерживает очередь заданий. Обе функции, `send` и `send-off`, возвращают управление немедленно, после постановки заданий в очередь, которые выполняются последовательно, в порядке их «передачи», в потоках, специально выделенных для выполнения заданий агентов. Результат каждого задания устанавливается как новое состояние агента.

Единственное различие между функциями `send` и `send-off` – типы передаваемых им заданий. Задания, помещаемые в очередь с помощью `send`, выполняются с помощью пула потоков фиксированного размера, который настраивается так, чтобы не превысить текущие аппаратные возможности[1]. По этой причине функция `send` *никогда не должна использоваться для передачи заданий, которые могут выполнять операции ввода/вывода или другие блокирующие операции*, по крайней мере, если блокирующие задания могут препятствовать другим, неблокирующим, вычислительным заданиям полностью использовать этот вычислительные мощности.

---

[1] Например, в системе с двухъядерным процессором для `send` будет создан пул, содержащий не более четырех потоков, в системе с четырехъядерным процессором будет создан пул с восемью потоками, и так далее.

Задания, передаваемые с помощью функции `send-off`, напротив, обрабатываются неограниченным пулом потоков (по совпадению тем же самым, который обеспечивает выполнение объектов `future`), что позволяет параллельно выполнять любое количество потенциально блокирующих заданий, не связанных с массивными вычислениями.

Теперь мы можем в общих чертах представить, как действуют агенты:

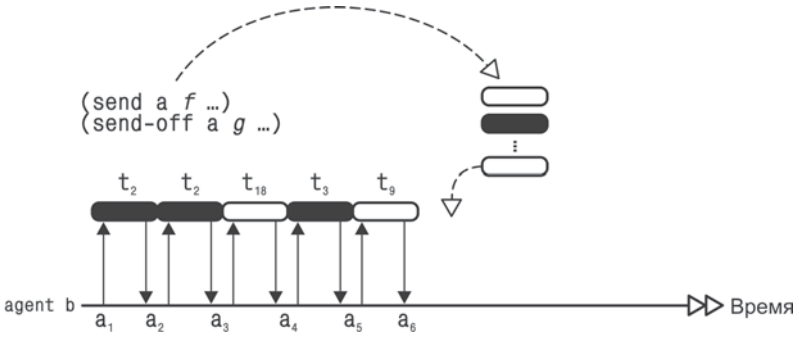

Рис. 4.12. Постановка в очередь и выполнение заданий агентов, изменяющих их состояния

Задания помещаются в очередь агента с помощью `send` или `send-off` (на рис. 4.12 представлены блоками разного цвета). Агент передает в эти задания свое текущее состояние, выполняя вычисления в потоке из пула, связанного с функцией, использовавшейся для постановки заданий в очередь. Таким образом, если предположить, что задания, представленные на рис. 4.12 черными блоками выполняют массивные вычисления, тогда потоки t_2 и t_3 принадлежат пулу фиксированного размера, связанному с функцией `send`, а потоки t_9 и t_{18} – пулу неограниченного размера, связанному с функцией `send-off`. Значения, возвращаемые заданиями, становятся новым состоянием агента.

Несмотря на тонкости, присущие семантике агентов, использовать их необычайно просто:

```
(def a (agent 500))                    ❶
;= #'user/a
(send a range 1000)                    ❷
;= #<Agent@53d2f8be: 500>
@a
;= (500 501 502 503 504 ... 999)
```

Агенты не соответствуют потокам выполнения

Между количеством агентов, созданных программой, и количеством действующих потоков выполнения, обслуживающих этих агентов, нет прямой связи. Оно в большей степени определяется количеством конкурирующих заданий, поставленных в очереди с помощью `send-off`: так как это – единственные задания, обслуживаемые неограниченным пулом потоков, для их выполнения может запрашиваться создание новых системных потоков выполнения. Не забывайте, что задания для каждого отдельно взятого агента выполняются последовательно, поэтому, чтобы создать 100 потоков выполнения (например), необходимо создать как минимум 100 агентов, для каждого из которого задания определены с помощью `send-off`.

Это подразумевает следующее:

1. Количество агентов ограничивается только объемом доступной памяти[1].
2. Количество конкурирующих заданий, добавляемых в очередь с помощью функции `send`, фактически ничем не ограничивается, однако количество потоков, выделенных для их обработки строго определено, поэтому снова здесь все зависит только от объема доступной памяти.

❶ Создается агент с начальным значением `500`.

❷ С помощью `send` устанавливается задание для агента, состоящее из функции `range` и дополнительного аргумента `1000`. В другом потоке выполнения агент получит значение, равное результату выражения `(range @a 1000)`.

Обе функции, `send` и `send-off`, возвращают переданного им агента. При установке задания в оболочке REPL вполне возможно, что вы увидите результат вычислений немедленно, выполнив операцию вывода агента. В зависимости от сложности задания и как быстро оно может быть поставлено в очередь, оно может завершиться уже к моменту, когда REPL получит шанс вывести агента, возвращаемого функцией `send` или `send-off`:

```
(def a (agent 0))
;= #'user/a
(send a inc)
;= #<Agent@65f7bb1f: 1>
```

С другой стороны, может возникнуть ситуация, когда продолжение работы возможно только после получения результатов выпол-

[1] Без изменения параметров управления динамической памятью (heap) можно создать много тысяч агентов, а выполнив дополнительные настройки JVM – миллионы.

нения задания, и приходится опрашивать агента, в ожидании окончания вычислений. Чтобы упростить задачу, можно заблокировать выполнение текущего потока на ожидании готовности агента с помощью `await`[1]:

```
(def a (agent 5000))
(def b (agent 10000))

(send-off a #(Thread/sleep %))
;= #<Agent@da7d7b5: 5000>
(send-off b #(Thread/sleep %))
;= #<Agent@c0cd75b: 10000>
@a                                    ❶
;= 5000
(await a b)                           ❷
;= nil
@a                                    ❸
;= nil
```

❶ На выполнение функции, переданной агенту `a`, потребуется пять секунд, поэтому его значение к этому моменту еще не обновилось.

❷ С помощью `await` текущий поток выполнения блокируется в ожидании выполнения всех заданий, переданных агенту. В данном случае вызов заблокируется примерно на 10 секунд, потому что именно это время требуется на выполнение функции, переданной агенту `b`.

❸ Когда функция `await` вернет управление, задания будут выполнены и значение агента обновится. Обратите внимание, что другое задание может изменить значение `a` до того, как вы сможете разыменовать его!

Функция `await-for` делает то же самое, но позволяет указать предельное время ожидания.

Обработка ошибок в заданиях агентов

Агенты выполняют свои задания асинхронно, поэтому исключения, сгенерированные в ходе выполнения, не могут быть переданы в поток, инициировавший ошибочное задание. По умолчанию, столкнувшись с ошибкой, агент просто прерывает выполнение: вы все еще можете получить последнее его состояние, но последующие задания невозможно будет поставить в очередь:

---

[1] В зависимости от особенностей реализации, которая может измениться в будущем, текущий размер очереди заданий агента `someagent` можно определять, вызывая метод `(.getQueueCount someagent)`.

```
(def a (agent nil))
;= #'user/a
(send a (fn [_] (throw (Exception. "something is wrong"))))
;= #<Agent@3cf71b00: nil>
a
;= #<Agent@3cf71b00 FAILED: nil>
(send a identity)                                                    ❶
;= #<Exception java.lang.Exception: something is wrong>
```

❶ Попытка добавить задание в агент, где возникла ошибка, возбудит исключение. Если потребуется явно проверить наличие ошибки, используйте функцию `agent-error`, которая вернет исключение или `nil`, если агент находится в работоспособном состоянии.

Восстановить работоспособность агента после ошибки можно с помощью функции `restart-agent`, которая повторно инициализирует агента указанным значением и разрешит ему вновь принимать задания. С помощью необязательного флага `:clear-actions` в вызове `restart-agent` можно удалить из очереди все невыполненные задания. В противном случае агент немедленно продолжит их выполнение.

```
(restart-agent a 42)
;= 42
(send a inc)                                                         ❶
;= #<Agent@5f2308c9: 43>
(reduce send a (for [x (range 3)]                                    ❷
                 (fn [_] (throw (Exception. (str "error #" x))))))
;= #<Agent@5f2308c9: 43>
(agent-error a)
;= #<Exception java.lang.Exception: error #0>
(restart-agent a 42)
;= 42
(agent-error a)                                                      ❸
;= #<Exception java.lang.Exception: error #1>
(restart-agent a 42 :clear-actions true)                             ❹
;= 42
 (agent-error a)
;= nil
```

❶ Перезапуск агента восстановит его работоспособность после ошибки и позволит ему принимать новые задания.

❷ Однако, если очередь агента содержит другие задания, которые в дальнейшем вызовут ошибку...

❸ ...функция `restart-agent` должна вызываться отдельно для каждого такого задания.

❹ Передача параметра `:clear-actions` в вызов `restart-agent` приведет к очистке очереди перед восстановлением работоспособности агента, и гарантирует, что никакие другие задания в очереди, обреченные на ошибку, не прервут выполнение агента немедленно.

Это – режим обработки ошибок по умолчанию, когда агент оказывается в неработоспособном состоянии и требует восстановления. Он особенно удобен, когда можно вручную вмешаться в работу агента, например, в оболочке REPL[1]. Более гибкий и потенциально более удобный режим обработки ошибок можно установить, изменив настройки по умолчанию агентов.

Режимы и обработчики ошибок в агентах

Режим по умолчанию, когда ошибка в агенте переводит его в неработоспособное состояние – один из двух поддерживаемых режимов. Функция `agent` принимает параметр `:error-mode` со значением `:fail` (по умолчанию) или `:continue`[2]; агент, действующий в режиме `:continue`, просто игнорирует ошибки, возникающие в ходе выполнения заданий, и принимает новые задания без лишних сложностей:

```
(def a (agent nil :error-mode :continue))
;= #'user/a
(send a (fn [_] (throw (Exception. "something is wrong"))))
;= #<Agent@44a5b703: nil>
(send a identity)
;= #<Agent@44a5b703: nil>
```

Это делает ненужной функцию `restart-agent`, однако игнорирование ошибок по умолчанию без возможности вмешаться в их обработку, в общем случае не самая лучшая идея. Поэтому режим `:continue` реакции на ошибки практически всегда используется в паре с обработчиком ошибок – функцией, принимающей два аргумента (агента и ошибку) и вызываемой всякий раз, когда агент генерирует ис-

---

[1] То есть, с помощью оболочки REPL, подключенной к вашему окружению; см. раздел «Отладка, мониторинг и исправление программ в REPL во время эксплуатации» в главе 10.

[2] Изменить режим реагирования агента на ошибки можно с помощью `set-error-mode!`.

ключение. Указать обработчик ошибок можно при создании агента, передав его в параметре :error-handler[1]:

```
(def a (agent nil
              :error-mode :continue
              :error-handler (fn [the-agent exception]
                               (.println System/out (.getMessage exception)))))
;= #'user/a
(send a (fn [_] (throw (Exception. "something is wrong"))))
;= #<Agent@bb07c59: nil>
; something is wrong
(send a identity)
:= #<Agent@bb07c59: nil>
```

Разумеется, в функции :error-handler можно выполнять операции намного более сложные, чем простой вывод в консоль информации об исключении: чтобы избежать дальнейших ошибок можно изменить некоторые данные в приложении, можно повторить выполнение задания или переключить параметр :error-mode агента обратно в состояние :fail, если известно, что это единственный безопасный способ прервать выполнение заданий:

```
(set-error-handler! a (fn [the-agent exception]
                        (when (= "FATAL" (.getMessage exception))
                          (set-error-mode! the-agent :fail))))
;= nil
(send a (fn [_] (throw (Exception. "FATAL"))))
;= #<Agent@6fe546fd: nil>
(send a identity)
;= #<Exception java.lang.Exception: FATAL>
```

Ввод/вывод, транзакции и вложенная передача заданий

В отличие от ссылок и атомов, агенты можно смело использовать для координации ввода/вывода и выполнения других блокирующих операций. Это придает им особую практическую ценность в любом приложении, использующем ссылки и транзакционную память для управления изменением своего состояния с течением времени. Кро-

[1] Установить обработчик ошибок после создания агента можно с помощью функции set-error-handler!.

ме того, благодаря особенностям семантики, агенты часто являются идеальным инструментом, упрощающим асинхронные операции ввода/вывода, даже когда ссылки вообще не участвуют в работе.

Агенты выполняют свои задания по порядку, поэтому они обеспечивают естественную синхронизацию операций с побочными эффектами. Агента можно настроить так, что его состояние будет играть роль дескриптора некоторого контекста – `OutputStream` для файла или сетевого сокета, соединение с базой данных, канал связи с очередью сообщений и так далее – и быть уверенными, что все задания, передаваемые агенту, будут иметь исключительный доступ к этому контексту на все время их выполнения. Это упрощает интеграцию элементов экосистемы Clojure, включая ссылки атомы, обычно используемые для минимизации побочных эффектов, с остальным миром.

Кого-то может волновать вопрос, касающийся возможности использования агентов в рамках транзакций STM. Передача задания агенту – это операция с побочным эффектом и, казалось бы, она подвержена непредсказуемым эффектам, связанным с перезапуском транзакций или другими операциями, имеющими побочные эффекты, такими как изменение состояний атомов или запись в файл. К счастью это не так.

Агенты интегрированы в реализацию поддержки транзакционной памяти STM так, что задания, передаваемые агентам с помощью функций `send` и `send-off` из транзакций, удерживаются от выполнения, пока транзакция не будет благополучно подтверждена. То есть, даже если транзакция будет повторена 100 раз, задание будет передано агенту только один раз, и все задания, переданные в процессе выполнения транзакции, будут помещены в очередь сразу после благополучного завершения транзакции. То же относится и к вызовам `send` и `send-off` из заданий агентов, называемых *вложенными передачами заданий*. Они также будут удерживаться от передачи в очередь до завершения задания. В обоих случаях отправляемые задания, ожидающие завершения родительского задания или транзакции STM, могут быть полностью ликвидированы, если функция-валидатор прервет операцию изменения состояния.

Для иллюстрации описанной семантики и демонстрации ее преимуществ рассмотрим пару примеров, в которых используются агенты для упрощения координации операций ввода/вывода, а также ссылки и механизм транзакционной памяти, и являющиеся частями высоконагруженных систем ввода/вывода с параллельной обработкой данных.

Сохранение состояний ссылок в журнале на основе агента

В игре, разрабатывавшейся в разделе «Механика изменения ссылок», выше, для поддержки состояний персонажей в многопользовательском конкурентном окружении использовались ссылки и демонстрировались возможности механизма транзакционной памяти в подобных окружениях. Однако любая игра подобная этой, особенно многопользовательская, должна следить за действиями игрока, сохранять информацию об этих действиях и их влиянии на персонажи. При этом не хотелось бы добавлять какую-либо поддержку журналирования или сохранения информации, как и другие операции ввода/вывода, в ядро игрового движка: любая сохраненная информация сама может стать недействительной из-за повторов транзакций.

Простейший способ решить эту проблему – реализовать отложенную запись информации с помощью функций-наблюдателей (watchers) и агентов. Для начала определим агентов, которые будут осуществлять вывод. В данном примере предполагается, что все подобные агенты будут содержать реализацию `java.io.Writers`, Java-интерфейса, определяющего API потоков вывода для персонажей:

```
(require '[clojure.java.io :as io])

(def console (agent *out*))
(def character-log (agent (io/writer "character-states.log" :append true)))
```

Один из этих агентов содержит `*out*` (который сам является экземпляром `Writer`), другой – экземпляр `Writer`, сливающий информацию в файл *character-states.log*, находящийся в текущем каталоге. Эти экземпляры `Writer` будут выводить содержимое, передаваемое им заданием `write`:

```
(defn write
  [^java.io.Writer w & content]
  (doseq [x (interpose " " content)]
    (.write w (str x)))
  (doto w
    (.write "\n")
    .flush))
```

Функция `write` принимает в первом аргументе экземпляр `Writer` (состояние агента, в очередь которого будет помещаться задание)

и произвольное количество других значений для записи. Она записывает значения, разделяя их пробелами, затем выводит знак перевода строки, после чего выталкивает содержимое `Writer`, чтобы оно фактически было записано на диск или выведено в консоль, а не хранилось в буферах экземпляров `Writer`.

Наконец, нам нужна функция, добавляющая функцию-наблюдателя к произвольному ссылочному типу, которая будет использоваться для соединения ссылок, представляющих персонажи, с агентами, хранящими экземпляры `Writer`:

```
(defn log-reference
  [reference & writer-agents]
  (add-watch reference :log
             (fn [_ reference old new]
               (doseq [writer-agent writer-agents]
                 (send-off writer-agent write new)))))
```

При каждом изменении состояния ссылки, ее новое состояние вместе с функцией `write` будет посылаться каждому агенту, переданному функции `log-reference`. Теперь нам осталось лишь добавить функцию-наблюдателя в персонажи, для которых требуется регистрировать изменение состояния, и начать битву:

```
(def smaug (character "Smaug" :health 500 :strength 400))
(def bilbo (character "Bilbo" :health 100 :strength 100))
(def gandalf (character "Gandalf" :health 75 :mana 1000))

(log-reference bilbo console character-log)
(log-reference smaug console character-log)

(wait-futures 1
              (play bilbo attack smaug)
              (play smaug attack bilbo)
              (play gandalf heal bilbo))

;{:max-health 500,:strength 400,:name "Smaug",:items #{},:health 490.052618}
;{:max-health 100,:strength 100,:name "Bilbo",:items #{},:health 61.5012391}
;{:max-health 100,:strength 100,:name "Bilbo",:items #{},:health 100.0}      ❶
;{:max-health 100,:strength 100,:name "Bilbo",:items #{},:health 67.3425151}
;{:max-health 100,:strength 100,:name "Bilbo",:items #{},:health 100.0}
;{:max-health 500,:strength 400,:name "Smaug",:items #{},:health 480.990141}
; ...
```

❶ Здесь можно наблюдать эффект оздоравливающего воздействия Гендальфа на Бильбо при каждом сохранении значения `:health` в журнале.

Та же информация будет записана в файл *character-states.log*. Важно отметить, что мы сохраняем информацию в файл и выводим ее в консоль, только потому, что они являются наиболее доступными – этот же подход с успехом можно использовать для сохранения информации в базе данных, очереди сообщений и так далее.

Использование функции-наблюдателя, как в данном случае, дает нам возможность сохранять информацию об изменении состояния каждого персонажа (например, в файле или в базе данных) не вторгаясь в функции, реализующие эти изменения.

Чтобы обеспечить сохранение информации, получаемой внутри транзакции, такой как количество нападений и оздоровлений, кто кому и что сделал, и так далее – нужно всего лишь предусмотреть передачу задания `write` нашим агентам в теле любой функции, выполняющей изменения, которые требуется зарегистрировать:

```
(defn attack
  [aggressor target]
  (dosync
    (let [damage (* (rand 0.1) (:strength @aggressor) (ensure daylight))]
      (send-off console write
        (:name @aggressor) "hits" (:name @target) "for" damage)
      (commute target update-in [:health] #(max 0 (- % damage))))))

(defn heal
   [healer target]
   (dosync
     (let [aid (min (* (rand 0.1) (:mana @healer))
                    (- (:max-health @target) (:health @target)))]
       (when (pos? aid)
         (send-off console write
           (:name @healer) "heals" (:name @target) "for" aid)
         (commute healer update-in [:mana] - (max 5 (/ aid 5)))
         (alter target update-in [:health] + aid)))))

(dosync
  (alter smaug assoc :health 500)
  (alter bilbo assoc :health 100))
; {:max-health 100, :strength 100, :name "Bilbo", :items #{}, :health 100}
```

```
; {:max-health 500, :strength 400, :name "Smaug", :items #{}, :health 500}

(wait-futures 1
              (play bilbo attack smaug)
              (play smaug attack bilbo)
              (play gandalf heal bilbo))
;{:max-health 500,:strength 400,:name "Smaug",:items #{},:health 497.414581}
; Bilbo hits Smaug for 2.585418463393845
;{:max-health 100,:strength 100,:name "Bilbo",:items #{},:health 66.6262521}
; Smaug hits Bilbo for 33.373747881474934
;{:max-health 500,:strength 400,:name "Smaug",:items #{},:health 494.667477}
; Bilbo hits Smaug for 2.747103668676348
;{:max-health 100,:strength 100,:name "Bilbo",:items #{},:health 100.0}
; Gandalf heals Bilbo for 33.37374788147494
; ...
```

В результате объединения всего этого со ссылками, представляющими персонажей, получился автоматический механизм сохранения информации, который смело можно использовать совместно с атомами и транзакциями, допускающими возможность повторения. Мы реализовали вывод в консоль и в файл, чтобы сохранить пример максимально простым, но точно также можно было бы реализовать сохранение информации об изменениях в базе данных. В любом случае, данный пример демонстрирует не только приемы использования атомов и ссылок, но и возможность совместного использования ресурсов ввода/вывода в конкурентной среде без применения низкоуровневых блокировок и не рискуя допустить ошибку при управлении ими.

Использование агентов для распределения нагрузки

На первый взгляд кажется неудобным, да и ненужным разделять задания для агентов на две категории. Однако без разделения на блокирующие и не блокирующие задания агенты могут потерять способность эффективно использовать ресурсы, необходимые для обслуживания нагрузки разного рода – центральный процессор, дисковый ввод/вывод, пропускная способность сетевого канала и так далее.

Например, допустим, что наше приложение предназначено для обработки очереди сообщений. Операция чтения сообщений наверняка будет блокироваться в ожидании поступления данных из сети, если очередь не предназначена для обмена сообщениями внутри

процесса и зависит от семантики ожидания поступления обрабатываемых данных. Однако обработка каждого отдельного сообщения, как правило, является вычислительной задачей.

По описанию очень похоже на поискового веб-робота. Агенты позволяют довольно легко создавать такие программы гибкими и эффективными. Здесь мы напишем очень простую программу[1], но она демонстрирует, как можно использовать агенты для управления нагрузкой и ее распределения.

Для начала создадим несколько основных функций для обработки содержимого веб-страниц. Функция `links-from` принимает базовый адрес URL с соответствующей HTML-страницей и возвращает последовательность ссылок, найденных в этой странице. Функция `words-from` принимает некоторый код разметки HTML и извлекает из нее текст, возвращая последовательность найденных слов, в которых все знаки приведены к нижнему регистру:

```
(require '[net.cgrand.enlive-html :as enlive])
(use '[clojure.string :only (lower-case)])
(import '(java.net URL MalformedURLException))

(defn- links-from
  [base-url html]
  (remove nil? (for [link (enlive/select html [:a])]
                 (when-let [href (-> link :attrs :href)]
                   (try
                     (URL. base-url href)
                     ; игнорировать недействительные адреса URL
                     (catch MalformedURLException e))))))

(defn- words-from
  [html]
  (let [chunks (-> html
                 (enlive/at [:script] nil)
                 (enlive/select [:body enlive/text-node]))]
    (->> chunks
      (mapcat (partial re-seq #"\w+"))
      (remove (partial re-matches #"\d+"))
      (map lower-case))))
```

[1] И не самую благонравную, так как она способна поглотить значительную часть пропускной способности сетевого соединения, что считается неприемлемым для веб-роботов. Приносим наши извинения компании BBC за злоупотребление их добротой при создании примера!

Для работы с разметкой HTML здесь используется библиотека Enlive, которая подробно будет обсуждаться в разделе «Enlive: преобразование HTML с применением селекторов», в главе 16, но эти подробности нас пока не интересуют, наша задача – с помощью агентов максимально эффективно использовать все ресурсы для обработки веб-содержимого.

Наш веб-робот имеет три пула состояний:

1. Первый – очередь Java с поддержкой использования в многопоточной среде, будет хранить еще необработанные адреса URL. Эта очередь будет называться `url-queue`. Далее, в каждой извлеченной странице мы будем...
2. Отыскивать все ссылки, чтобы позднее выполнить их обход. Эти ссылки будут добавляться в множество, хранящееся внутри атома с именем `crawled-urls`, а еще не посещавшиеся адреса URL будут добавляться в очередь `url-queue`. Наконец...
3. Мы будем извлекать весь текст из каждой страницы и подсчитывать, сколько раз встретилось каждое слово. Суммы будут храниться в ассоциативном массиве, отображающем слова в их количества, который в свою очередь будет содержаться в атоме с именем `word-freqs`:

```
(def url-queue (LinkedBlockingQueue.))
(def crawled-urls (atom #{}))
(def word-freqs (atom {}))
```

Мы также создадим группу агентов с целью максимально использовать все доступные ресурсы[1], но нам необходимо подумать, какие состояния будут храниться и какие операции будут использоваться для перехода от одного состояния к другому. Во многих случаях полезно представлять состояние агента и операции, применяемые к нему, как конечный автомат (машина, с конечным числом состояний); мы уже знаем, как будет действовать наш веб-робот, но нам необходимо формализовать его.

Состояние агента в каждой точке процесса должно быть очевидным: прежде чем извлечь содержимое по адресу URL, агент должен

[1] Еще один недостаток нашего робота: в большинстве случаев в веб-роботах для поддержки состояний в памяти используются очередь сообщений и подходящая база данных. К счастью, это не затрагивает семантику нашего примера, который относительно легко можно адаптировать для использования всего перечисленного.

иметь этот адрес URL (или некоторый источник адресами); прежде чем анализировать страницу, агент должен получить ее содержимое; а прежде чем обновлять счетчики слов, должны иметься результаты анализа страницы. Так как в нашем случае состояний не так много, мы можем упростить задачу, позволив всем операциям, реализующим переходы, определять следующую операцию (переход) которую следует применить к агенту.

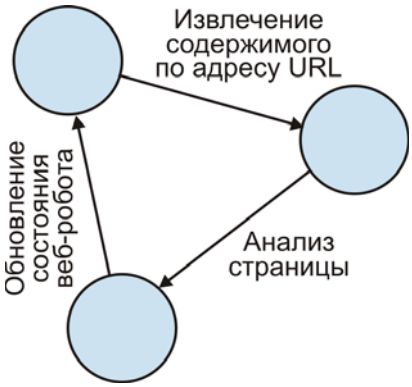

Рис. 4.13. Основная диаграмма изменения состояния веб-робота

Чтобы увидеть, на что это похоже, определим нашу группу агентов. Их начальное состояние, соответствующее состоянию перед переходом «Извлечение содержимого по адресу URL» на рис. 4.13, является ассоциативным массивом, содержащим очередь, откуда будет извлекаться следующий адрес URL, как и сам следующий переход – функция с именем `get-url`:

```
(declare get-url)

(def agents
     (set (repeatedly 25 #(agent {::t #'get-url :queue url-queue}))))
```
❶

❶ Состояние нашего агента всегда будет иметь слот `::t`[1], содержащий функцию, реализующую следующий переход[2].

---

[1] Мы используем ключевое слово с пространством имен, чтобы избежать потенциальных конфликтов имен с другими частями состояния, которые могут добавлены в агенты, если данная реализация веб-робота когда-нибудь перерастет границы собственного пространства имен.

[2] В зависимости от диапазона состояний, которые будут хранить ваши агенты, передача им мультиметода (multimethod) или функции протокола может оказаться более изящным и эффективным средством реализации переходов между множеством разных состояний агента. Подробнее о мультиметодах и протоколах будет рассказываться в главе 7 и в разделе «Протоколы», в главе 6, соответственно.

Три перехода, изображенные на рис. 4.13, реализованы в виде трех заданий агентов: `get-url`, `process` и `handle-results`.

Функция `get-url` будет ожидать получения адреса URL из очереди (напомню, что каждый агент изначально имеет исходное значение `url-queue`). Она сохранит в состоянии агента ассоциативный массив с адресами URL, извлеченными из очереди, и их содержимым:

```
(declare run process handle-results)                    ❶

(defn ^::blocking get-url
  [{:keys [^BlockingQueue queue] :as state}]
  (let [url (as-url (.take queue))]
    (try
      (if (@crawled-urls url)                           ❷
        state
        {:url url
         :content (slurp url)
         ::t #'process})
      (catch Exception e
        ;; пропустить URL, вызвавший ошибку при загрузке содержимого
        state)
      (finally (run *agent*)))))
```

❶ Функция `run` будет представлена и описана чуть ниже.

❷ Если адрес URL, извлеченный из очереди, уже посещался (или при попытке загрузить его содержимое возникла ошибка), состояние агента остается без изменений. Эта особенность реализации нашего конечного автомата добавляет цикл, в котором функция `get-url` иногда будет вызываться несколько раз подряд, прежде чем произойдет переход к другому состоянию.

Функция `process` проанализирует содержимое, извлеченное по адресу URL, с помощью `links-from` и `words-from` получит ссылки и создаст ассоциативный массив, содержащий счетчики всех слов, найденных в содержимом. Она сохранит в состоянии агента ассоциативный массив с этими значениями, а также URL содержимого:

```
(defn process
  [{:keys [url content]}]
  (try
    (let [html (enlive/html-resource (java.io.StringReader. content))]
      {::t #'handle-results
       :url url
```

```
        :links (links-from url html)
        :words (reduce (fn [m word]
                         (update-in m [word] (fnil inc 0)))      ❶
                       {}
                       (words-from html))})
    (finally (run *agent*))))
```

❶ Ассоциативный массив `:words` содержит все найденные слова с их счетчиками, который создается в результате свертки последовательности этих слов. `fnil` – это функция высшего порядка, возвращающая новую функцию, которая вызывает функцию, переданную `fnil` и подставляет значение по умолчанию (здесь, 0) вместо любого аргумента со значением `nil`. Это избавляет нас от необходимости явно проверять значения в ассоциативном массиве слов на равенство `nil`, и возвращать 1 в этом случае.

Функция `handle-results` обновит три ключевые составляющие состояния: добавит только что проанализированный URL в `crawled-urls`, поместит все вновь найденные ссылки в `url-queue` и объединит ассоциативный массив со счетчиками слов с накопительным ассоциативным массивом `word-freqs`. Функция `handle-results` возвращает ассоциативный массив с состоянием, содержащий `url-queue` и переход `get-url`, оставив агента в исходном состоянии.

```
(defn ^::blocking handle-results
  [{:keys [url links words]}]
  (try
    (swap! crawled-urls conj url)
    (doseq [url links]
      (.put url-queue url))
    (swap! word-freqs (partial merge-with +) words)

    {::t #'get-url :queue url-queue}
    (finally (run *agent*))))
```

Можно заметить, что все функции, используемые в качестве заданий агента, содержат форму `try` с формой `clause`, содержащей единственный вызов `run` с аргументом `*agent*`[1]. Мы нигде не определили имя `*agent*`. Обычно для несвязанного имени Clojure создает пере-

[1] Этот шаблон повторяется в трех функциях, однако совсем несложно было бы написать макрос, избавляющий от шаблонного кода.

менную, которая во время выполнения задания агента будет связана с текущим агентом. То есть, выражение `(run *agent*)` в каждом из представленных выше заданий, вызывает `run` с единственным аргументом – агентом, выполняющим задание.

Это – распространенная идиома, используемая при работе с агентами, позволяющая организовать непрерывное их выполнение. В случае с нашим веб-роботом, `run` – это функция, которая добавляет в очередь очередную функцию перехода, получая ее из слота `::t` состояния агента. Но, в каждом задании заранее известно, какой переход должен быть выполнен следующим, зачем добавлять этот лишний уровень косвенности в виде вызова `run`? На то есть две причины:

1. Вполне оправданно ожидать, что каждая функция, используемая в качестве задания агента, знает, какой следующий переход должен быть выполнен, согласно возвращаемому ею новому состоянию, но в ней невозможно определить, будет ли следующее задание выполнять блокирующие операции или нет. Эти хлопоты лучше оставить на долю реализации самих переходов (и их информированных авторов), Функция `run` будет проверять присутствие (или отсутствие) слота `::blocking` в метаданных каждого перехода и выбирать, какую функцию использовать, `send` или `send-off`, для передачи функции перехода[1].
2. Функция `run` может проверить, был ли приостановлен агент. Это условие определяется наличием логически истинного значения `::paused` в метаданных агента.

Пример 4.8. Функция run – «главный цикл» веб-робота

```
(defn paused? [agent] (::paused (meta agent)))

(defn run
  ([] (doseq [a agents] (run a)))
  ([a]
    (when (agents a)
      (send a (fn [{transition ::t :as state}]
                (when-not (paused? *agent*)
```

[1] Необходимость использовать метаданные отчасти объясняет, почему мы задействовали переменные для хранения функций переходов вместо самих функций. Кроме того, применение переменных позволяет упростить дальнейшее развитие веб-робота; подробнее о причинах рассказывается в разделе «Ограничения при переопределении конструкций», в главе 10.

```
                (let [dispatch-fn (if (-> transition meta ::blocking)
                                    send-off
                                    send)]
                  (dispatch-fn *agent* transition)))
              state)))))
```

При вызове без аргументов функция `run` запускает все (не приостановленные) агенты.

Возможность приостановки играет важную роль, так как у нас должен быть некоторый рычаг, позволяющий остановить работу веб-робота без грубого вмешательства. С помощью метаданных можно указать функции `run`, что она не должна передавать следующее задание агенту, а функции `pause` и `restart` дают нам возможность приостановить или возобновить выполнение заданий агентами простым изменением их метаданных:

```
(defn pause
  ([] (doseq [a agents] (pause a)))
  ([a] (alter-meta! a assoc ::paused true)))

(defn restart
  ([] (doseq [a agents] (restart a)))
  ([a]
    (alter-meta! a dissoc ::paused)
    (run a)))
```

Теперь можно попробовать выполнить обход нескольких веб-страниц! Было бы желательно иметь возможность повторно запускать веб-робота с пустым исходным состоянием, поэтому будет удобно иметь тестовую функцию, очищающую состояние веб-робота. Функция `test-crawler` выполняет такую очистку, а также добавляет начальный адрес URL в `url-queue` и позволяет агенту работать всего 60 секунд, благодаря чему можно приблизительно оценить производительность:

```
(defn test-crawler
  "Сбрасывает веб-робота в исходное состояние, добавляет указанный URL
   в url-queue и запускает робота на 60 секунд. Возвращает вектор,
   содержащий количество просмотренных URL и число URL, накопленных
   в процессе обхода, которые еще не были посещены."
  [agent-count starting-url]
  (def agents (set (repeatedly agent-count
```

```
                    #(agent {::t #'get-url :queue url-queue}))))  ❶
  (.clear url-queue)
  (swap! crawled-urls empty)
  (swap! word-freqs empty)
  (.add url-queue starting-url)
  (run)
  (Thread/sleep 60000)
  (pause)
  [(count @crawled-urls) (count url-queue)])
```

❶ В разделе «Переменные в языке Clojure не являются переменными в классическом понимании», выше, мы предупреждали вас об опасности переопределения переменных в теле функции, но это один из редких случаев, когда такой прием оправдан: функция, которая никогда не вызывается, кроме как в оболочке REPL и используемая исключительно для тестирования.

Для начала попробуем задействовать одного агента, и в качестве начальной страницы используем страницу новостей BBC:

```
(test-crawler 1 "http://www.bbc.co.uk/news/")
;= [86 14598]
```

За минуту было проанализировано 86 страниц. Разумеется, это не самый лучший показатель. Попробуем задействовать 25 агентов, которые одновременно будут выполнять и блокирующие задания, выполняющие загрузку страниц, и вычислительные – выполняющие анализ страниц и обработку содержащегося в них текста:

```
(test-crawler 25 "http://www.bbc.co.uk/news/")
;= [670 81775]
```

Неплохо! За 60 секунд было посещено 670 страниц. Прирост производительности на порядок удалось получить всего лишь за счет увеличения числа агентов[1].

Посмотрим полученные значения частот встречаемости слов. Мы легко можем извлечь наиболее часто и наиболее редко встречающиеся слова с их частотами:

---

[1] Разумеется, у вас могут получиться другие результаты, так как они сильно зависят от мощности процессора и скорости подключения к Интернету. Однако относительный прирост производительности при увеличении количества агентов с 1 до 25 должен быть сопоставимым с нашим.

```
(->> (sort-by val @word-freqs)
  reverse
  (take 10))
;= (["the" 23083] ["to" 14308] ["of" 11243] ["bbc" 10969] ["in" 9473]
;=  ["a" 9214] ["and" 8595] ["for" 5203] ["is" 4844] ["on" 4364])
(->> (sort-by val @word-freqs)
  (take 10))
;= (["relieved" 1] ["karim" 1] ["gnome" 1] ["brummell" 1] ["mccredie" 1]
;=  ["ensinar" 1] ["estrictas" 1] ["arap" 1] ["forcibly" 1] ["kitchin" 1])
```

Нам удалось создать функционирующего веб-робота, выполняющего некоторую работу. Конечно, робот получился не самый оптимальный – как уже говорилось, это простейшая реализация, требующая дальнейшего расширения, но основные принципы должны быть понятны.

Теперь вспомните, как выше в этом разделе говорилось, что разделение заданий агентов на те, которые могут блокироваться (например, в ожидании ввода/вывода), и которые не могут (то есть, сугубо вычислительные задания), позволяет максимально эффективно использовать все ресурсы, имеющиеся в нашем распоряжении. Мы можем проверить это утверждение, например, пометив функцию `process`, как блокирующую операцию, и гарантировав тем самым, что она всегда будет передаваться агентам с помощью `send-off`, то есть, в пул потоков неограниченного размера:

```
(alter-meta! #'process assoc ::blocking true)
;= {:arglists ([{:keys [url content]}]), :ns #<Namespace user>,
;=  :name process, :user/blocking true}
```

В результате этого парсинг разметки HTML, поиск ссылок и подсчет количества слов в тексте будут выполняться с применением неограниченного количества потоков.

```
(test-crawler 25 "http://www.bbc.co.uk/news/")
;= [573 80576]
```

Фактически такой шаг отрицательно сказался на пропускной способности – общее снижение составило около 15 процентов. Теперь за ядра процессора конкурируют 25 активных (и голодных) агентов, и эта конкуренция привела к снижению рабочих показателей.

Механизмы параллельного выполнения в Java

Теперь, когда мы так подробно познакомились с многочисленными механизмами параллельного выполнения и управления состоянием в языке Clojure, следует заметить, что «родная» поддержка многопоточности в языке Java, простые механизмы блокировки и библиотеки – особенно пакеты `java.util.concurrent.*` – с успехом можно использовать и в Clojure. В частности, последние весьма широко используются в реализации механизмов параллельного выполнения в самом языке Clojure, но он не предоставляет оберток для них, поэтому, чтобы задействовать эти механизмы в своих приложениях, необходимо познакомиться с особенностями их использования.

Мы еще не занимались исследованием всей механики взаимодействий между Clojure и Java – мы займемся этим в главе 9 – но примеры, которые будут представлены здесь, достаточно просты, чтобы понять их, не имея глубоких познаний в этой области.

В языке Java имеется два основных интерфейса, `java.lang.Runnable` и `java.util.concurrent.Callable`, которые в языке Clojure реализуются всеми функциями без параметров. Это означает, что функции без параметров на языке Clojure можно передавать любым Java API, требующим объект, реализующий один из этих интерфейсов, включая `Thread`:

```
(.start (Thread. #(println "Running...")))
;= Running...
;= nil
```

Пакеты `java.util.concurrent.*` предлагают массу возможностей параллельного выполнения, используемых в реализации собственных механизмов языка Clojure, многие из которых позволяют получить дополнительные преимущества в определенных ситуациях. В разделе «Использование агентов для распределения нагрузки», выше, уже демонстрировалось применение поточно-ориентированной очереди, `LinkedBlockingQueue`. Существуют и другие подобные типы с небольшими, но важными отличиями в семантике и производительности. Кроме того, имеется возможность организации потоков в пулы, поточно-ориентированные структуры данных (лучшая замена простым типам, таким как `java.util.HashMap`, на случай, если в программе на Clojure придется использовать изменяемые на месте структуры дан-

ных совместно с некоторым кодом на Java) и специализированные объекты, такие как `CountDownLatch`, позволяющие блокировать выполнение потока (объекта `future` или задания агента, переданного с помощью `send-off`), пока не произойдет некоторое количество событий.

Если вам интересно узнать, как пользоваться всеми этими возможностями и получить полное представление о низкоуровневых механизмах конкуренции в JVM, обращайтесь к книге «Java Concurrency in Practice» Брайен Гетц (Brian Goetz) и др.

Блокировки

Даже при наличии всех (более надежных) примитивов параллельного выполнения в языке Clojure, иногда все еще бывает необходимо пользоваться простейшими блокировками, особенно при работе с изменяемыми типами данных Java, такими как массивы. Конечно, приняв такое решение, вы остаетесь наедине с собой: вы не сможете больше полагаться на определенную семантику, гарантируемую этими примитивами. Но, как бы то ни было, вы всегда можете воспользоваться макросом `locking` для получения и удержания блокировки для заданного объекта на протяжении всего времени выполнения тела формы `locking`.

Итак, следующий код на языке Clojure:

```
(defn add
  [some-list value]
  (locking some-list
  (.add some-list value)))
```

эквивалентен коду на Java, Ruby и Python, соответственно:

```
// Java
public static void add (java.util.List someList, Object value) {
     synchronized (someList) {
          someList.add(value);
     }
}

# Ruby
require 'thread'
m = Mutex.new

def add (list, value)
```

```
    m.synchronize do
        list << value
    end
end

# Python
import threading
lock = threading.Lock()
def add (list, value):
    lock.acquire()
    list.append(value)
    lock.release()
```

В заключение

Конкурентное программирование – сложная задача, и многие популярные языки программирования делают ее еще сложнее. Обладая ясным разделением идентичности и состояния, свойственным неизменяемости, и надежными встроенными конструкциями для параллельного программирования, Clojure далеко ушел в этом направлении, сделав параллельное программирование проще и доступнее.

Часть II
СОЗДАНИЕ АБСТРАКЦИЙ

Глава 5. Макросы

Исторически, диалекты Lisp называют «программируемыми языками программирования». Язык Clojure полностью соответствует этому определению и, во многом благодаря его макросам. Макросы позволяют программисту расширять язык Clojure способами, сложными или невозможными в других языках.

Язык программирования – это средство создания абстракций. Вместо того, чтобы долго выполнять утомительную работу, программист может написать код один раз и использовать его как многоразовый модуль. Код может выполняться многократно в цикле. Код можно оформить в виде модуля, дать ему имя и использовать как функцию. Один и тот же код, включающий условные инструкции, может выполнять разные операции при разных обстоятельствах.

Совершенно очевидно, что некоторые языки предлагают более мощные средства абстракции, чем другие. Представьте на мгновение язык программирования, не имеющий циклов. Таким языком может быть и можно пользоваться, но разворачивать циклы вручную будет чрезвычайно утомительно. То же относится и к языку, не поддерживающему функций – возможно, он позволит делать то же самое, что и любой другой Тьюринг-полный язык, но код на таком языке придется повторять снова и снова.

Проще говоря, когда язык испытывает нехватку средств абстракции, возникает потребность писать массу шаблонного и повторяющегося кода, что является признаком существенной слабости такого языка. Макросы – мощное средство, потому что они дают возможность определять совершенно новые уровни абстракции внутри самого языка. Макросы являются важнейшим инструментом устранения шаблонного кода и выращивания языка программирования под потребности программиста.

Что такое макрос?

Макросы позволяют управлять компилятором Clojure. Они позволяют произвести тонкую настройку синтаксиса или полностью перевернуть с ног физические законы языка. Если Java – это «C++, из которого убрали все пистолеты, ножи и дубинки»[1], такие языки, как Ruby и Python, содержат в себе некоторый арсенал, хотя и ограниченный, то макросы в Clojure позволяют сконструировать любое оружие, и оно будет выглядеть так, как будто было встроено в язык изначально.

Ключом к пониманию макросов является полное представление о различиях между *временем выполнения* и *временем компиляции*.

Как вы уже знаете из раздела «Механизм чтения» главы 1, механизм чтения преобразует исходный программный код на языке Clojure из текстового представления в структуры данных, практически такие же, что вы используете в своей программе. Например, строка `"(foo [bar] :baz 123)"` преобразуется в список, содержащий символ, вектор с символом, ключевое слово и целое число. Это свойство, когда программный код представлен в виде собственных структур данных языка, называется *гомоиконностью* и является важным условием поддержки макросов[2].

После этого структуры данных обычно *вычисляются* или *выполняются*. При вычислении каждого типа данных используются определенные правила:

- многие литералы соответствуют сами себе (например, целые числа, строки, ключевые слова, векторы);
- символы разрешаются в значения соответствующих переменных из некоторого пространства имен;
- списки обозначают вызовы функций, специальных форм или макросов.

В промежутке между *чтением* и *выполнением*, когда производится компиляция, макросы занимают привилегированное положение в сравнении с функциями. Вызовы функций в исходном коде переносятся в скомпилированное представление этого кода. Они получают готовые аргументы в виде параметров и возвращают некоторый

---

[1] Эта цитата часто приписывается Джеймсу Гослингу (James Gosling), первоначальному архитектору языка Java.

[2] Более подробно о гомоиконности рассказывается в разделе «Гомоиконность», в главе 1.

результат на этапе выполнения, *Макросы, напротив, вызываются компилятором*, получают не скомпилированные структуры данных в виде аргументов и должны вернуть структуру данных на языке Clojure, которую можно скомпилировать. Например, если допустить, что `foo` – это функция, тогда выражение

```
(foo a b)
```

будет скомпилировано в вызов функции `foo` времени выполнения с двумя значениями, соответствующими именам `a` и `b`. Если предположить, что `bar` – это макрос

```
(bar a b)
```

тогда `bar` будет вызван компилятором Clojure с двумя аргументами, символами `a` и `b`, а не с их значениями[1]. Макрос `bar` может вычислить эти символы в соответствии с правилами языка, или реализовать собственную семантику, и может использовать для этого все возможности языка Clojure, а также все данные и функции, объявленные к данному моменту: макросы не ограничены некоторым подмножеством языка. В конечном итоге макрос `bar` должен вернуть компилятору структуру данных Clojure, которую можно подставить на его место. Это – рекурсивный процесс, потому что макрос может вернуть структуру данных, включающую другой макрос; и продолжается, пока не будет выполнен вызов последнего макроса.

Макросы являются инструментами создания абстракций. Вызов макроса обычно производит некоторый код, более объемный, чем сам вызов макроса. Поэтому данный процесс замены вызова макроса сгенерированным им программным кодом называется *развертыванием макроса* (macroexpansion). Как впервые говорилось в разделе «Интерактивная оболочка REPL для Clojure» главы 1, программный код на языке Clojure всегда компилируется, даже в оболочке REPL, и процесс развертывания макросов является важной и неотъемлемой частью процесса компиляции.

---

[1] Компилятор Clojure знает, что вызовы макросов следует обрабатывать иначе, чем вызовы функций или специальных форм благодаря особенностям реализации макросов, которые сами по себе являются функциями с дополнительными метаданными, идентифицирующими их как макросы. Убедиться в этом можно, например, исследовав метаданные макроса `var`, такие как `(meta #'or)`.

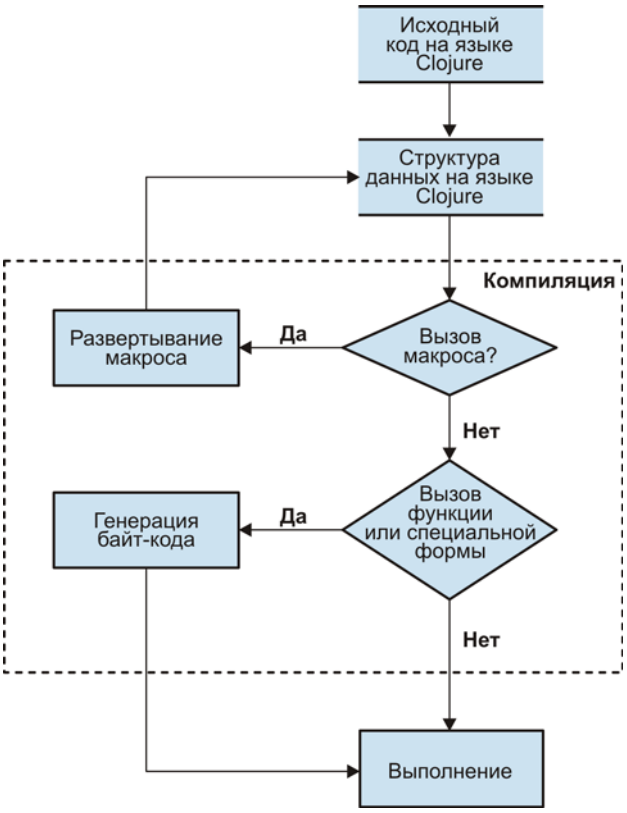

Рис. 5.1. Модель компиляции Clojure

Примечание. Компилятор гарантирует замену всех вызовов макросов их развернутыми представлениями задолго до наступления этапа выполнения; то есть макросы всегда выполняются только на этапе компиляции.

Чем не являются макросы

Способность писать код, манипулирующий кодом, не является уникальной особенностью языка Clojure или Lisp. Однако не все системы, поддерживающие ее, обладают одинаковыми возможностями.

Например, в языке C имеется препроцессор, который на этапе компиляции выполняет замену одного исходного кода в текстовом представлении другим. Такие текстовые системы поддержки макро-

сов обладают меньшими возможностями, чем макросы в стиле языка Lisp, потому что работают со строками, а не со структурированными данными, представляющими код. Некоторые из этих недостатков проявляются также в механизмах выполнения программного кода в текстовом представлении, таких как `eval` в языке Ruby, который мы будем сравнивать с макросами Clojure в разделе «Сравнение макросов и механизма eval в Ruby», ниже.

Аналогично, *генераторы кода* нельзя назвать эквивалентными макросам. Обычно они принимают некоторое высокоуровневое представление, например, формальную грамматику или описание объектной модели, и создают код, реализующий ее. Такие системы имеют определенную практическую ценность, однако их недостатками являются: выделение компиляции в отдельный этап (тогда как в Clojure обработка макросов, как и любого другого программного кода, выполняется на этапе компиляции), использование моделей данных (тогда как в Clojure макросы являются обычными структурами данных) и невозможность комбинирования (тогда как в Clojure макросы легко могут комбинироваться друг с другом).

Наконец, существует множество языков, обеспечивающих API для доступа к компилятору, позволяя изменять код, написанный на этих языках. Примерами могут служить: процессоры аннотаций в Java, построители абстрактных синтаксических деревьев в Groovy, Template Haskell и расширения (plug-ins) компилятора Scala. Это очень мощные системы, похожие на поддержку макросов в Clojure и позволяющие создавать синтаксические абстракции. Несмотря на различия, самым удивительным является то, что все эти системы по определению предоставляют доступ к внутреннему API и модели данных языка, которые обычно далеко спрятаны о исходного кода. Clojure, будучи гомоиконным языком, делает интерфейс и модель макросов такой же простой, как обычные структуры данных.

Что могут макросы, чего не могут функции?

В первый момент трудно оценить всю мощь и полезность макросов, поэтому рассмотрим простой пример, который поможет в этом.

Первая версия Java была выпущена в 1996. Восемь лет спустя, в версию Java 5 были добавлены полезные и долгожданные расширенные циклы `for`, позволяющие заменить чрезвычайно многословный код, как показано ниже:

```
for (int i = 0; i < collection.size(); i++) {
SomeType var = (SomeType)collection.get(i);
    ...
}
```

более краткой альтернативой:

```
for (SomeType var : collection) {
    ...
}
```

Такого рода изменения в языке играют важную роль. Расширенный цикл `for` в Java позволил избавиться от лишних сложностей и ошибок: используя расширенный цикл `for` невозможно выйти за границы массива или по неосторожности начать итерации с элемента с индексом 1 вместо 0.

Как программисты, мы можем взглянуть на конкретный пример использования расширенного синтаксиса `for` и механически преобразовать его в эквивалентный цикл с использованием старого синтаксиса `for`. Единственными важными элементами здесь являются имена переменной цикла и коллекции, по которой выполняются итерации. Все остальное – изменение индексов, проверка границ и тому подобное – всего лишь шаблонный код.

Так почему никто не догадался добавить в Java расширенный цикл `for` раньше? Проблема в том, что в Java отсутствуют выразительные средства, которые позволили бы это. Расширенный цикл `for` не может быть реализован, как вызов обычного метода, а вызовы методов – единственный инструмент, имеющийся в вашем распоряжении в Java. Методы не могут привязывать значения к локальным переменным, находящимся за пределами самих методов. Методы не могут обеспечить условное вычисление своих аргументов – их аргументы всегда вычисляются и результаты передаются методам. Конечно, в Java можно было бы создать нечто, своим поведением похожее на расширенные циклы `for`, но это нечто определенно не будет выглядеть как то, что появилось в Java 5.

Чтобы добавить расширенный цикл `for`, необходимо внести изменения на уровне компилятора, а средний пользователь не обладает требуемыми для этого знаниями или возможностями. Как же обходились разработчики на Java без такой полезной особенности целых восемь лет? Они просто жили без нее.

В Clojure, напротив, *любой* программист за пару минут сможет написать несколько строк непривилегированного кода, реализующего макрос, который добавит императивную конструкцию цикла, напоминающую расширенный цикл `for` в Java 5[1]:

```
(defmacro foreach [[sym coll] & body]
  `(loop [coll# ~coll]
     (when-let [[~sym & xs#] (seq coll#)]
       ~@body
       (recur xs#))))
;= #'user/foreach
(foreach [x [1 2 3]]
  (println x))
; 1
; 2
; 3
```

Функции – отличный инструмент абстракции, но кое-что они просто не в состоянии делать, потому что вызываются во время выполнения и не имеют доступа к компилятору. В данном случае невозможно включить программный код (вызов `println`) в конструкцию цикла, используя только функции. Проще говоря, макросы дают программисту возможность добавлять в язык новые языковые конструкции.

Встроенные (built-in) конструкции и макросы. В сравнении со многими языками, Clojure обладает весьма ограниченным множеством встроенных операторов, называемых *специальными формами*. Напомню, что полный перечень (см. раздел «Специальные формы» в главе 1) содержит всего 16 элементов.

В этом списке отсутствует многое из того, что можно было бы ожидать найти там. Где конструкции циклов, такие как `while`, `for` и `doseq`? Конструкции определения, такие как `defn`, `defmacro`, `defrecord`? Условные конструкции, такие как `when`, `cond` и `condp`? Все то, чем мы пользуемся ежедневно, создавая программы на языке Clojure.

Возможно, кто-то удивится, узнав, что все они являются макросами. Если бы их не существовало вообще, вы легко могли бы напи-

[1] Макрос `foreach` приводится здесь исключительно в иллюстративных целях. В Clojure уже имеется надмножество функциональных возможностей, предлагаемых расширенным императивным циклом в Java, реализованное в виде `doseq`, в дополнение к его функциональной паре `for`.

сать их на Clojure, не вступая в контакт с авторами языка. В этом отношении Clojure резко контрастирует со многими другими языками, где мысли о создании собственного цикла или условной конструкции является чистой фантазией. Это также распространяется и на узкоспециализированные прикладные или предметные требования, поскольку небольшие изменения в синтаксисе способны сделать язык более подходящим для решения определенных задач. Таким образом, макросы стирают грань между «встроенными» (built in) и пользовательскими конструкциями, потому что последние получают тот же статус, что и первые, и различия между ними теряют свое значение.

Сравнение макросов и механизма eval в Ruby

На первый взгляд макросы могут показаться похожими на функцию `eval` в Ruby[1]. `eval` в Ruby – это встроенная (built-in) функция, которая выполняет программный код на этапе выполнения. В Ruby также имеются функции `class_eval` и `instance_eval`, выполняющие код в разных контекстах.

На языке Ruby можно написать следующий код:

```
x = 123
code = "puts VAR"
code.sub!(/VAR/, 'x')
eval code
```

И он выведет 123. Здесь сначала создается строка с программным кодом, затем она изменяется и выполняется.

Одно из важнейших отличий между макросами и функцией `eval` состоит в том, что `eval` компилирует код в строке во время выполнения программы. Это означает, например, что ошибки, которые обычно обнаруживаются на этапе компиляции, будут обнаружены в таком коде только во время выполнения. Взгляните:

```
code = <<END
  def foo
    puts "foo! # Упс! Забыли закрывающую кавычку
```

[1] А также в Python, JavaScript, PHP, Perl или в любом другом языке, позволяющем выполнять строки, содержащие программный код.

```
  end
END

if(rand(2) == 0)
  eval code
end
```

В половине случаев этот код будет компилироваться и работать без ошибок, потому что строка с кодом никогда не будет выполняться. В другой половине случаев будет возникать ошибка времени выполнения.

Напротив, макросы в Clojure компилируются на этапе компиляции. Подобная ошибка в Clojure будет обнаружена немедленно, потому что макросы – это обычный программный код на языке Clojure и попытка использовать недопустимый синтаксис будет оканчиваться неудачей механизма чтения. Например, следующий код даже не будет скомпилирован, не говоря уже о выполнении:

```
(defmacro foo []
  `(if (= 0 (rand-int 2))
     (println "foo!")))      ;; Упс! Забыли закрывающую кавычку
;= #<Exception java.lang.Exception: EOF while reading string>
```

Другое различие между макросами и функцией `eval` в Ruby проявляется сразу же, при попытке изменить код. Функция `eval` в Ruby принимает программный код в виде строки, не имеющей определенной структуры. Поэтому для изменения кода мы можем использовать только инструменты, предназначенные для работы со строками: регулярные выражения, а также операции конкатенации строк и извлечения подстроки.

Макросы в Clojure, напротив, оперируют не аморфными строками. Поскольку Clojure обладает гомоиконностью, макросы могут оперировать структурами данных, такими как списки, векторы, символы и так далее.

Изменение исходного кода в строке чревато ошибками. И эти ошибки не удастся обнаружить, пока не будет предпринята попытка выполнить код, что делает весь этот процесс достаточно опасным. Поэтому в мире Ruby часто можно услышать справедливые громкие протесты против использования `eval`.

Так, следующий код в мире Ruby посчитали бы весьма дурным тоном:

```
>> def print_sym(x)
>>   code = "p(" + x + ".to_sym)"
>> end
nil
>> eval print_sym "\"foo\""
:foo
nil
```

хотя его эквивалент в языке Clojure не вызывает таких нареканий:

```
(defmacro print-keyword [x]
  `(println (keyword ~x)))
;= #'user/print-keyword
(print-keyword "foo")
; :foo
;= nil
```

Пишем свой первый макрос

Давайте напишем простенький макрос, который будет делать то, что функциям не под силу. Допустим, что с целью озадачить своих коллег, мы решили записать все символы в своем коде задом наперед. То есть, наша цель в том, чтобы реализовать возможность писать такой код:

```
(reverse-it (nltnirp "foo"))
```

который при компиляции преобразовывался бы в такой:

```
(println "foo")
```

Совершенно очевидно, что это невозможно в таких языках, как Java, без изменения парсера или компилятора Java. Но мы легко можем добиться этого в Clojure, благодаря его макросам.

Наш макрос `reverse-it` должен переворачивать исходный код, который он получает в форме структур данных языка Clojure. То есть, на нужно отыскать все символы, извлечь строковые имена символов, перевернуть эти строки и вернуть новые символы на их места.

В пакете `clojure.walk` имеется удобная функция `postwalk`, позволяющая выполнять рекурсивный обход последовательности вложенных списков и выполнять операции с отдельными элементами. Это полностью соответствует нашим требованиям:

Пример 5.1. reverse-it, макрос переворачивания символов

```
(require '(clojure [string :as str]
                   [walk :as walk]))

(defmacro reverse-it
  [form]                                                    ❶
  (walk/postwalk #(if (symbol? %)                           ❷
                    (symbol (str/reverse (name %)))         ❸
                    %)
                 form))
```

❶ Наш макрос принимает единственный аргумент, с именем `form`.
❷ Он использует функцию `postwalk`, чтобы рекурсивно применить указанную анонимную функцию ко всем элементам в форме.
❸ Эта анонимная функция замещает все символы в форме другими символами, с перевернутыми именами, а остальные элементы оставляет без изменений.

Теперь мы можем писать такие нелепицы[1]:

```
(reverse-it
  (qesod [gra (egnar 5)]
    (nltnirp (cni gra))))
; 1
; 2
; 3
; 4
; 5
;= nil
```

потому что после развертывания макроса компилятор Clojure увидит обычный программный код:

```
(macroexpand-1 '(reverse-it
                  (qesod [gra (egnar 5)]
                    (nltnirp (cni gra)))))
;= (doseq [arg (range 5)]
;= (println (inc arg)))
```

[1] Макрос `reverse-it` является отличной демонстрацией возможности выполнять синтаксические преобразования в макросах, однако пользоваться такой возможностью считается дурным тоном.

Чтобы увидеть, какой код макрос вернет компилятору Clojure, здесь была использована функция `macroexpand-1`. Семейство функций `macroexpand` — это основной инструмент тестирования и отладки макросов, о которой рассказывается в следующем разделе.

Отладка макросов

Общеизвестно, что макросы могут оказаться весьма сложными в отладке. Компилятор Clojure успешно отыскивает самые разнообразные ошибки, но необходимо проявлять особую осторожность, чтобы использовать всю мощь макросов и не попасть в их ловушки.

Взгляните, что произойдет, если попытаться сослаться на переменную, которая еще не определена. В функции это вызовет ошибку на этапе компиляции:

```
(defn oops [arg] (frobnicate arg))
;= #<CompilerException java.lang.Exception:
;=  Unable to resolve symbol: frobnicate in this context (NO_SOURCE_FILE:1)>
```

К сожалению, если определить похожий макрос, ошибка уже не будет обнаруживаться компилятором:

```
(defmacro oops [arg] `(frobnicate ~arg))
;= #'user/oops
```

Однако ошибка обнаружится, если попытаться использовать макрос:

```
(oops 123)
;= #<CompilerException java.lang.IllegalStateException:
;=   Var user/frobnicate is unbound. (NO_SOURCE_FILE:0)>
```

Что же произошло? Напомню, что макросы выполняются на *этапе компиляции*. Во время компиляции Clojure не знает (да и не может знать), будет ли символ `frobnicate` ссылаться на переменную, имеющую во время выполнения некоторое значение. Макрос видит и возвращает только списки, символы и другие структуры данных. Макросу не известно, будут ли эти символы действительными во время выполнения кода, созданного им. Это обстоятельство может существенно осложнить отладку макросов, однако у нас есть несколько удобных инструментов.

Функции развертывания макросов

Самым основным инструментом, используемым при отладке макросов, является функция `macroexpand-1`. Она принимает структуру данных (в отладочном контексте – форма макроса) и передает ее компилятору Clojure, чтобы получить код, представляющий, напомню, обычную структуру данных, который будет выполнен в том месте, где встречен макрос. Ниже можно видеть, что макрос `oops` вызывается с целым числом и возвращает список из двух элементов, символ `frobnicate` (ссылающийся на несуществующую переменную) и целочисленный аргумент:

```
(macroexpand-1 '(oops 123))
;= (user/frobnicate 123)
```

Функция `macroexpand-1` просто развертывает макрос один раз. Не забывайте, что развертывание макросов может происходить многократно, если макрос возвращает код, содержащий вызов другого макроса. Если макрос производит вызов другого макроса и вам потребуется продолжить развертывание до состояния, когда в форме не останется макросов, используйте функцию `macroexpand`.

Вследствие того, что многие возможности языка Clojure сами реализованы в виде макросов, функция `macroexpand-1` часто оказывается весьма полезной, позволяя избежать получения чересчур подробных результатов, трудных для изучения. Вернемся к макросу `reverse-it`, который мы написали в примере 5.1, теперь мы готовы увидеть разницу между функциями `macroexpand-1` и `macroexpand`:

```
(macroexpand-1 '(reverse-it
                  (qesod [gra (egnar 5)]
                    (nltnirp (cni gra)))))
;= (doseq [arg (range 5)]
;= (println (inc arg)))

(pprint (macroexpand '(reverse-it
                        (qesod [gra (egnar 5)]
                          (nltnirp (cni gra))))))
; (loop*
;  [seq_1647
;   (clojure.core/seq (range 5))
;   chunk_1648
;   nil
```

```
;    count_1649
;    (clojure.core/int 0)
;    i_1650
;    (clojure.core/int 0)]
;   (if
;    (clojure.core/< i_1650 count_1649)
;    (clojure.core/let
;     [arg (.nth chunk_1648 i_1650)]
;     (do (println (inc arg)))
;     (recur
;      seq_1647
;      chunk_1648
;      count_1649
;      (clojure.core/unchecked-inc i_1650)))
;    (clojure.core/when-let
;     [seq_1647 (clojure.core/seq seq_1647)]
;     (if
;      (clojure.core/chunked-seq? seq_1647)
;      (clojure.core/let
;       [c__3798__auto__ (clojure.core/chunk-first seq_1647)]
;       (recur
;        (clojure.core/chunk-rest seq_1647)
;        c__3798__auto__
;        (clojure.core/int (clojure.core/count c__3798__auto__))
;        (clojure.core/int 0)))
;      (clojure.core/let
;       [arg (clojure.core/first seq_1647)]
;       (do (println (inc arg)))
;       (recur
;        (clojure.core/next seq_1647)
;        nil
;        (clojure.core/int 0)
;        (clojure.core/int 0)))))))
```

Макрос `reverse-it` возвращает форму `doseq`, которая сама является макросом, опирающимся на специальную форму `loop`, разворачивающимся в довольно длинное представление. В большинстве случаев бывает достаточно увидеть только «верхний уровень» развертывания макроса, что делает `macroexpand-1` более предпочтительной.

Полное развертывание макросов. Ни `macroexpand`, ни `macroexpand-1` не развертывают вложенные формы. Например, ниже выполняется попытка применить `macroexpand` к форме `cond`, в результате которой выводится вызов `if` с веткой «else», содержащей неразвернутую форму `cond`:

```
(macroexpand '(cond a b c d))
;= (if a b (clojure.core/cond c d))
```

Возможность получить полностью развернутый макрос может оказаться весьма полезной. В большинстве случаев для этого можно использовать функцию `clojure.walk/macroexpand-all`:

```
(require '[clojure.walk :as w])

(w/macroexpand-all '(cond a b c d))
;= (if a b (if c d nil))
```

Функция `macroexpand-all` удобна, но в значительной степени она представляет собой простой хак, лишь частично имитируя полное развертывание макросов, выполняемое компилятором. Например, она недостаточно точно обрабатывает специальные формы:

```
(w/macroexpand-all ''(when x a))
;= (quote (if x (do a)))
```

Это выражение должно было развернуться в `(quote (when x a))` – развертывание не должно было пойти дальше специальной формы `quote`. Точно так же `macroexpand-all` не поддерживает неявные аргументы `&env` и `&form`, которые мы подробно обсудим в разделе «Неявные аргументы: &env и &form», ниже, вместе с улучшенным вариантом `macroexpand`, поддерживающим их.

Синтаксис

Поскольку макросы возвращают структуры данных Clojure, мы часто будем использовать списки для представления последующих вызовов функций, специальных форм или других макросов. Поэтому нам нужны инструменты для создания этих списков. Здесь вполне допустимо использовать простейший из инструментов – функцию `list`:

```
(defmacro hello
  [name]
  (list 'println name))

(macroexpand '(hello "Brian"))
;= (println "Brian")
```

Однако в более сложных макросах, возвращающих нечто большее, чем простой плоский список, применение этой функции быстро становится утомительным занятием. Ниже показано, как выглядит исходный код для стандартного макроса `while`, реализованный таким способом:

```
(defmacro while
  [test & body]
  (list 'loop []
    (concat (list 'when test) body)
      '(recur)))
```

Суть этого макроса затерялась среди вызовов `list` и `concat`.

Чтобы этого не происходило, в Clojure имеется синтаксический сахар для работы со списками и интерполяции в них именованных значений. Реализация макроса `while` с этим сахаром[1] выглядит гораздо понятнее:

```
(defmacro while
  [test & body]
  `(loop []
     (when ~test
       ~@body
      (recur))))
```

В этом макросе использованы три инструмента: `syntax-quote`[2], `unquote` и `unquote-splicing`. Этот синтаксис несвойственен макросам. Его можно использовать в функциях или в другом коде, но так как макросы часто и много работают со списками, наиболее часто вы будете использовать их именно в макросах[3].

---

[1] Представленный здесь код точно отражает реализацию макроса в стандартной библиотеке Clojure.

[2] В других диалектах Lisp этот механизм называют «обратной кавычкой» (backquote) или «квази-кавычкой» (quasi-quote). Термин «синтаксическая кавычка» (syntax-quote) был введен Ричем Хикки (Rich Hickey), чтобы обеспечить его отличие от упомянутых вариантов.

[3] Примеры использования `syntax-quote` и `unquote` вне макросов можно найти в разделе «Подробности: -> и ->>», ниже.

Сравнение quote и syntax-quote

К настоящему моменту вы должны быть знакомы со специальной формой `quote`, возвращающей свои аргументы в исходном виде. Выражение `(quote (a b))` можно записать в более кратком виде: `'(a b)`. Оба варианта преобразуются в список из двух элементов, символов `a` и `b`.

В Clojure имеется еще одна форма маскирования (quoting). Форма *синтаксического маскирования* (syntax-quoting) выглядит также, как форма маскирования, но в ней вместо кавычки используется обратный апостроф (`).

Ниже демонстрируются два отличия между формами `quote` и `syntax-quote`. Во-первых, последняя из них полностью квалифицирует неквалифицированные символы текущим пространством имен:

```
(def foo 123)
;= #'user/foo
[foo (quote foo) 'foo `foo]
;= [123 foo foo user/foo]
```

Результатом синтаксического маскирования `` `foo `` (последний элемент «foo» в векторе) является символ с именем `user/foo`, включающим пространство имен, потому что он был определен в пространстве имен `user`. В другом пространстве имен элемент `` `foo `` будет прочитан как символ, квалифицированный другим пространством имен.

```
(in-ns 'bar)
`foo
;= bar/foo
```

Однако, если символ включает пространство имен или известно, что он ссылается на переменную из другого пространства имен, синтаксическое маскирование квалифицирует символ соответствующим пространством имен.

```
(ns baz (:require [user :as u]))

`map
;= clojure.core/map
`u/foo
;= user/foo
`foo
;= baz/foo
```

Такая квалификация символов по умолчанию позволяет гарантировать, что макрос не сгенерирует код, который по оплошности сошлется или переопределит значение, уже имеющее имя в контексте, где оно используется. Это называется *гигиеной* макросов и обсуждается в разделе «Гигиена», ниже.

Второе отличие между маскированием и синтаксическим маскированием заключается в том, что синтаксическое маскирование позволяет *размаскировать* (unquoting): некоторые элементы в списке можно выборочно размаскировать, вызывая их вычисление в области видимости формы синтаксического маскирования.

unquote и unquote-splicing

При создании каркаса исходного кода в макросах часто приходится создавать списки, одни элементы которых требуется вычислить, а другие – нет. Один из способов сделать это (достаточно болезненный, если бы это был единственный выбор) заключается в использовании `list` и синтаксическом маскировании всех элементов, которые не должны вычисляться.

```
(list `map `println [foo])
;= (clojure.core/map clojure.core/println [123])
```

Более короткая и более удобная возможность заключается в том, чтобы маскировать весь список и *размаскировать* те элементы, которые должны быть вычислены на месте. Делается это с помощью ~.

```
`(map println [~foo])
;= (clojure.core/map clojure.core/println [123])

`(map println ~[foo])
;= (clojure.core/map clojure.core/println [123])
```

Отметьте, что в сравнении с версией, где явно используется `list`, форма синтаксического маскирования выглядит идентичной тому, как мы записывали формы размаскирования, с дополнительными ` и ~.

Обратите также внимание, что во втором примере размаскирование списка или вектора размаскирует форму целиком. Эту особенность можно использовать для выполнения вызовов функций внутри форм синтаксического маскирования:

```
`(println ~(keyword (str foo)))
;= (clojure.core/println :123)
```

Другой распространенный случай: начинать со списка форм и распаковывать в него содержимое другого списка. И снова, самая прямолинейная версия на основе `list` и `concat` выглядит достаточно неуклюже:

```
(let [defs '((def x 123)
             (def y 456))]
  (concat (list 'do) defs))
;= (do (def x 123) (def y 456))
```

Гораздо удобнее выглядит использование функции `unquote-splicing` (оператор `~@`), автоматически выполняющей объединение.

```
(let [defs '((def x 123)
             (def y 456))]
  `(do ~@defs))
;= (do (def x 123) (def y 456))
```

Здесь элементы из списка `defs` внедряются в синтаксически замаскированный список. Это – весьма распространенная идиома при создании макросов. Например, макросы, принимающие несколько форм в качестве «тела кода», часто выглядят так:

```
(defmacro foo
  [& body]
  `(do-something ~@body))

(macroexpand-1 '(foo (doseq [x (range 5)]
                       (println x))
                     :done))
;= (user/do-something
;=    (doseq [x (range 5)]
;=      (println x))
;=    :done)
```

Последний аргумент `body` хранит упорядоченную коллекцию параметров макроса, а последующее выражение `~@body` в макросе распаковывает этот список в окружающий контекст.

Функция `syntax-quote` в соединении с функциями `unquote` и `unquote-splicing` позволяет интерпретировать структуры данных языка Clojure – и, соответственно, код, который можно вернуть из макроса – как шаблоны: вы можете сформировать каркас формы с помощью `syntax-quote` и затем выборочно заполнять параметризованные части каркаса значениями и результатами выражений, получаемых с применением `unquote`.

Примечание. В `syntax-quote`, `unquote` и `unquote-splicing` нет ничего магического: это функции в стандартной библиотеке Clojure, которые подставляются механизмом чтения на место соответствующих им синтаксических конструкций `` ` ``, `~` и `~@`. Увидеть, что происходит за кулисами, можно, замаскировав синтаксически замаскированное выражение, чтобы подавить его вычисление:

```
``(map println ~[foo])
;= (clojure.core/seq
;=    (clojure.core/concat
;=      (clojure.core/list (quote clojure.core/map))
;=      (clojure.core/list (quote clojure.core/println))
;=      (clojure.core/list [foo])))
```

Когда следует использовать макросы

Макросы дают большую власть, а власть неизбежно влечет за собой ответственность и сопровождается целым списком предостережений.

Макросы действуют на этапе компиляции. То есть, они не являются частью выполняющейся программы на языке Clojure, подобно функциям. Макросы не имеют доступа к информации времени выполнения, такой как текущие значения переменных. Макросы видят только нескомпилированные структуры данных, полученные из исходных текстов.

Рассмотрим простую функцию, возвращающую приветствие. Ее можно написать как функцию или как макрос:

```
(defn fn-hello [x]
  (str "Hello, " x "!"))

(defmacro macro-hello [x]
  `(str "Hello, " ~x "!"))
```

В одних ситуациях они ведут себя одинаково:

```
(fn-hello "Brian")
;= "Hello, Brian!"
(macro-hello "Brian")
;= "Hello, Brian!"
```

Но в других – по-разному:

```
(map fn-hello ["Brian" "Not Brian"])
;= ("Hello, Brian!" "Hello, Not Brian!")
(map macro-hello ["Brian" "Not Brian"])
;= #<CompilerException java.lang.RuntimeException:
;=    Can't take value of a macro: #'user/macro-hello,
;=    compiling:(NO_SOURCE_PATH:1)>
```

В последнем случае при использовании макроса `macro-hello` возникла ошибка, потому что мы попытались применить значение макроса во время выполнения. Функционально макросы не существуют во время выполнения[1], они не могут комбинироваться или передаваться как значения, и поэтому бессмысленно пытаться отобразить макрос на значения в коллекции, как в примере выше.

Чтобы использовать макросы в подобных контекстах, в местах применения их необходимо заключать в форму `fn` или в литерал анонимной функции. Этот прием перенесет применение макроса обратно на этап компиляции, когда будет компилироваться включающая его функция. В общем случае это порождает некоторую неуклюжесть, когда передача значения функции выглядела бы проще и понятнее:

```
(map #(macro-hello %) ["Brian" "Not Brian"])
;= ("Hello, Brian!" "Hello, Not Brian!")
```

Вместо применения нескладных оберток `fn` или `#(...)` можно было бы попробовать написать еще один макрос – обертку для функции `map`. Однако этот путь напоминает извилистую нору – применение макроса порождает потребность добавлять другие макросы:

---

[1] Это не совсем верно, как описывается в разделе «Тестирование и отладка использования &env» (ниже), однако возможность использования макросов во время выполнения – довольно запутанная и неподдерживаемая особенность реализации, не стоящая затрачиваемых усилий.

чем больше макросов вы пишете, тем больше макросов требуется, чтобы решить проблему их недоступности во время выполнения. По этой причине макросы не пригодны для использования во многих идиомах функционального программирования, призывающих к использованию функций высшего порядка.

Из вышесказанного следуют два вывода: один незначительный, а другой важный. Во-первых, макросы являются удобным и мощным инструментом в одном контексте (на этапе компиляции), но могут осложнять жизнь в другом (на этапе выполнения), заставляя вычленять реализацию макроса и сохранять ее в отдельной функции. На этапе компиляции макрос можно использовать без лишних сложностей, но если его возможности потребуются там, где нужна простая функция, этот путь тоже доступен вам.

Во-вторых, что более важно, *макросы должны использоваться, только когда вам нужны будут собственные конструкции языка*. Отсюда следует, что они не должны использоваться там, где функции оказываются эффективнее. Макросы являются единственным решением, когда требуется:

- особая семантика вычислений;
- собственный синтаксис для часто используемых шаблонов или предметно-ориентированных форм записи;
- получить дополнительные выгоды от предварительной компиляции промежуточных данных.

С другой стороны, всегда следует спрашивать себя – нет ли более функционального (то есть, не требующего каких-то специальных правил вычисления и, соответственно, макросов) способа достижения той же цели.

Гигиена

Традиционно, одной из самых больших проблем, с которой сталкиваются при разработке макросов, является генерирование программного кода, некорректно взаимодействующего с другим программным кодом. В языке Clojure имеются средства защиты, отсутствующие в других диалектах Lisp, но вероятность появления ошибок все еще сохраняется.

Код, сгенерированный макросами часто внедряется в другой код, и часто содержит код, добавленный пользователем. В любом случае, некоторые символы наверняка будут связаны со значениями пользователем макроса. В макросе можно также определить собственные

привязки, вступающие в конфликт с внешним или внутренним контекстом пользовательского кода, передаваемого макросу, что может привести к появлению ошибок, очень сложных в обнаружении. Макросы, исключающие такого рода проблемы, называют *гигиеничными макросами* (hygienic macros).

Рассмотрим макрос, требующий значение для let-связки. Мы выбрали имя, ни о чем не говорящее пользователю макроса и невидимое для него, но мы должны были выбрать *некоторое* имя. По простоте душевной мы выбрали имя `x`:

```
(defmacro unhygienic
  [& body]
  `(let [x :oops]
     ~@body))
;= #'user/unhygenic
(unhygienic (println "x:" x))
;= #<CompilerException java.lang.RuntimeException:
;=    Can't let qualified name: user/x, compiling:(NO_SOURCE_PATH:1)>
```

Компилятор Clojure достаточно умен, чтобы скомпилировать такой код. Как описывалось в разделе «Сравнение quote и syntax-quote», выше, форма синтаксического маскирования (`syntax-quote`) квалифицирует все неквалифицированные символы соответствующим пространством имен. Убедиться в этом можно, если выполнить развертывание нашего макроса:

```
(macroexpand-1 `(unhygienic (println "x:" x)))
;= (clojure.core/let [user/x :oops]
;=    (clojure.core/println "x:" user/x))
```

Ссылка на `x` была преобразована в ссылку на `user/x`, но форма `let` требует, чтобы имена для новых привязок являлись неквалифицированными символами, поэтому такой макрос всегда будет порождать код, дающий ошибку во время компиляции. Можно было бы некорректно «устранить» эту проблему, воспользовавшись инструментами маскирования и размаскирования:

```
(defmacro still-unhygienic
  [& body]
  `(let [~'x :oops]                    ❶
     ~@body))
```

```
;= #'user/still-unhygenic
(still-unhygienic (println "x:" x))
; x: :oops
;= nil
(macroexpand-1 '(still-unhygienic
                     (println "x:" x)))
;= (clojure.core/let [x :oops]
;= (println "x:" x))
```

❶ В выражении ~'x используется форма размаскирования (~), чтобы принудительно использовать неквалифицированный символ x в качестве имени значения в let-связке.

Это позволяет хотя бы запустить код, но появилась другая крупная ошибка. Проблема в том, что имя x, которое определяется макросом в локальной области видимости, может конфликтовать с локальными привязками внутри или снаружи кода, генерируемого макросом. Взгляните, как легко нарушить работу этого макроса:

```
(let [x :this-is-important]
  (still-unhygienic
    (println "x:" x)))
; x: :oops
```

Здесь мы уже используем имя x для другого локального значения, но форма let, порождаемая макросом, «по-тихому» сохраняет в x собственное значение. Пользователь макроса может никогда не узнать об этом, не прочитав исходный код. К счастью Clojure имеет простой механизм предотвращения конфликтов имен, таких как в данном коде, генерируемом макросом, а его использование требует меньше усилий, чем предыдущее неправильное «решение».

Генераторы символов во спасение

При определении привязок в макросах желательно было бы иметь возможность динамически генерировать имена, которые никогда не будут конфликтовать с именами за пределами области видимости макроса или внутри кода, передаваемого макросу в качестве аргумента. К счастью в Clojure имеется механизм и простой синтаксис для создания таких имен: *генератор символов* (gensyms). Функция gensym возвращает символ, гарантированно являющийся уникальным. При каждом вызове функции возвращается новый символ.

```
(gensym)
;= G__2386
(gensym)
;= G__2391
```

gensym также принимает аргумент: строку, которая будет использоваться как префикс в сгенерированном символе.

```
(gensym "sym")
;= sym2396
(gensym "sym")
;= sym2402
```

Функцию gensym можно использовать всегда, когда вам потребуется уникальный символ, но главное ее назначение – оказывать помощь в создании гигиеничных макросов:

```
(defmacro hygienic
  [& body]
  (let [sym (gensym)]                ❶
    `(let [~sym :macro-value]        ❷
       ~@body)))
;= #'user/hygienic
(let [x :important-value]
  (hygienic (println "x:" x)))
; x: :important-value
;= nil
```

❶ Теперь, вместо явно указанного имени для новой локальной привязки используется гарантированно уникальный символ, возвращаемый функцией gensym...

❷ ...и syntax-unquote, чтобы вставить его в код, возвращаемый макросом.

Теперь код безопасен. Имя x, выбранное для *нашей* локальной привязки, больше не будет конфликтовать с локальной привязкой, генерируемой макросом, потому что в ней используется имя, сгенерированное функцией gensym.

Генераторы символов используются в макросах настолько часто, что был введен более краткий синтаксис их использования. Любой символ, оканчивающийся знаком # внутри формы syntax-quote, автоматически будет развернут в динамически сгенерированный символ, и будет развертываться в один и тот же символ при каждом вхождении. Этот синтаксис называется *автоматическим генератором сим-*

волов (auto-gensym). Следующий код эквивалентен предыдущему определению макроса `hygienic`:

```
(defmacro hygienic
  [& body]
  `(let [x# :macro-value]
     ~@body))
```

Символ `x#` в форме синтаксического маскирования при развертывании макроса будет преобразован в нечто, подобное `x__3507__auto__`.

В пределах единственной формы синтаксического маскирования, все вхождения указанного динамически генерируемого символа будут преобразовываться в один и тот же фактический символ:

```
`(x# x#)
;= (x__1447__auto__ x__1447__auto__)
```

Это позволяет в одной форме синтаксического маскирования многократно использовать один и тот же динамический символ, с легко опознаваемым именем в пределах макроса.

```
(defmacro auto-gensyms
  [& numbers]
  `(let [x# (rand-int 10)]                    ❶
     (+ x# ~@numbers)))                       ❷
;= #'user/auto-gensyms
(auto-gensyms 1 2 3 4 5)
;= 22
(macroexpand-1 '(auto-gensyms 1 2 3 4 5))    ❸
;= (clojure.core/let [x__570__auto__ (clojure.core/rand-int 10)]
;= (clojure.core/+ x__570__auto__ 1 2 3 4 5))
```

❶ Для создания локальной привязки в коде, передаваемом макросу, используется автоматически сгенерированный символ `x#`.

❷ То же имя `x#` используется для ссылки на локальную привязку.

❸ Развернув макрос, можно убедиться, что каждое обращение к автоматически сгенерированному символу замещается уникальным символом, гарантирующим отсутствие конфликтов с другими именованными привязками.

Однако, имейте в виду, что при использовании механизма автоматического создания символов, фактические символы будут совпадать, только в пределах одной формы синтаксического маскирования:

```
[`x# `x#]
;= [x__1450__auto__ x__1451__auto__]
```

Это означает, что реализация `doto`, представленная ниже, которая использует один и тот же автоматически генерируемый символ в разных формах синтаксического маскирования,

```
(defmacro our-doto [expr & forms]
  `(let [obj# ~expr]                              ❶
     ~@(map (fn [[f & args]]
              `(~f obj# ~@args)) forms)           ❷
     obj#))
```

❶ Первый раз символ `obj#` встречается в одной форме синтаксического маскирования.

❷ Второй раз символ `obj#` встречается в другой форме синтаксического маскирования, внутри формы `unquote-splicing`. В результате создаются два разных фактических символа, хотя при создании макроса предполагалось, что в каждом случае будет использоваться одно и то же значение.

будет терпеть неудачу:

```
(our-doto "It works"
  (println "I can't believe it"))
;= #<CompilerException java.lang.RuntimeException:
;=    Unable to resolve symbol: obj__1456__auto__ in this context,
;=    compiling:(NO_SOURCE_PATH:1)>
```

потому что два динамических символа `obj#` находятся в области видимости разных форм синтаксического маскирования. В подобных случаях необходимо опять вернуться к явному использованию `gensym`:

```
(defmacro our-doto [expr & forms]
  (let [obj (gensym "obj")]
    `(let [~obj ~expr]
       ~@(map (fn [[f & args]]
                `(~f ~obj ~@args)) forms)   ❶
       ~obj)))
```

❶ В этом месте синтаксическое маскирование в значительной степени бессмысленно и его следует заменить на `(list* f obj args)`.

И тогда все заработает:

```
(our-doto "It works"
  (println "I can't believe it")
  (println "I still can't believe it"))
; It works I can't believe it
; It works I still can't believe it
;= "It works"
```

Предоставление пользователю права выбора имен

Если в макросе потребуется выполнить привязку, которая *должна быть* доступна в коде, вызывающем макрос, нам потребуется выбрать имя для этой привязки и при этом побеспокоиться о гигиене.

Можно было бы просто выбрать некоторое фиксированное имя и указать в документации, что макрос всегда использует это имя. Это не самое удачное решение, но иногда оно применяется. Например, если макрос развертывается в код, взаимодействующий с Java, вы могли бы использовать в нем фиксированное имя `this`[1].Макрос, который преднамеренно использует фиксированные имена, называется *анафорическим* (anaphoric)[2]. Обычно принято избегать получения такой характеристики, потому что пользователи таких макросов должны постоянно помнить, какие символы неявно используются в качестве имен в локальных привязках в их области видимости.

Проще и разумнее дать пользователю возможность выбрать символ(ы) для привязки. Так как макрос не вычисляет свои аргументы, пользователь легко может передать символ макросу, который должен использоваться в сгенерированном коде:

```
(defmacro with
  [name & body]
  `(let [~name 5]
     ~@body))
```

---

[1] Такой прием используется в макросе `proxy`, но это редкое исключение из правила, согласно которому имена в Clojure всегда выбираются пользователем. Другим исключением является сам макрос `defmacro`! Смотрите обсуждение `&env` и `&form` далее в этой главе.

[1] Некоторые примеры анафорических макросов можно найти в разделе «Избавление от шаблонного кода», в главе 14.

```
;= #'user/with
(with bar (+ 10 bar))
;= 15
(with foo (+ 40 foo))
;= 45
```

Двукратное вычисление

Одной из распространенных и коварных проблем макросов является проблема *двукратного вычисления*. Коварная, потому что во многих случаях она остается незаметной. Двукратное вычисление выполняется, когда аргумент макроса появляется два или более раз в развернутом теле. Взгляните на макрос `spy`:

```
(defmacro spy [x]
  `(do
     (println "spied" '~x ~x)
     ~x))
```

Этот макрос выводит и возвращает значение указанного выражения. Он действует в соответствии с ожиданиями, когда `x` является константой. Но как только `x` превращается в выражение, требующее дорогостоящих вычислений, и/или порождает побочные эффекты, мы получим неожиданный результат:

```
(spy 2)
; spied 2 2
;= 2
(spy (rand-int 10))
; spied (rand-int 10) 9
;= 7
```

В первом случае проблема не наблюдается, потому что `2` является константой. Во втором случае выражение `(rand-int 10)` вычисляется дважды в коде, генерируемом макросом, каждый раз возвращая разные результаты. Такое поведение становится очевидным, если развернуть макрос:

```
(macroexpand-1 '(spy (rand-int 10)))
;= (do (println (rand-int 10))
;= (rand-int 10))
```

Эта может превратиться в очень *серьезную* проблему, если вместо `rand-int` использовать функцию `launch-missiles`[1]. ТЧтобы избавиться от этой проблемы, всегда следует использовать локальную привязку (обычно динамически сгенерированный символ), если при развертывании макроса происходит дублирование аргумента[2]:

```
(defmacro spy [x]
  `(let [x# ~x]
     (println "spied" '~x x#)
     x#))

(macroexpand-1 '(spy (rand-int 10)))
;= (let [x__725__auto__ (rand-int 10)]
;=   (println x__725__auto__ '(rand-int 10))
;=   x__725__auto__)
```

Этот прием гарантирует, что указанное выражение никогда не будет вычисляться более одного раза:

```
(spy (rand-int 10))
; spied (rand-int 10) 9
;= 9
```

Двукратное вычисление, даже если его можно обойти, – это *дурно пахнущий код* (code smell), что может служить признаком необходимости выделения некоторой части операций в отдельную функцию.

```
(defn spy-helper [expr value]
  (println expr value)
  value)

(defmacro spy [x]
  `(spy-helper '~x ~x))
```

В данном случае отпала необходимость вводить локальную привязку с автоматически генерируемым символом.

---

[1] Имя `launch-missiles` можно перевести как «запустить ракеты». – *Прим. перев.*

[2] Точнее, если аргумент макроса появляется более одного раза в одной и той же ветви макроса.

Распространенные идиомы и шаблоны макросов

Словосочетание *шаблоны макросов* может показаться оксюмороном, потому что макросы сами по себе призваны устранить последние оставшиеся шаблоны[1]. Вместо того, чтобы рассуждать здесь о шаблонном, повторяющемся коде, сместим фокус на проблему стиля и порассуждаем, как сделать макросы более идиоматичными с точки зрения языка Clojure.

Требуйте, чтобы новые локальные привязки определялись в виде вектора. Когда макрос вводит новую локальную область видимости со связанными именами, эти имена вместе с их значениями должны передаваться в виде вектора, обычно в первом аргументе макроса. Это поможет соответствовать идиоме, которой следуют многие основные формы и макросы в языке Clojure, включая `let`, `if-let`, `for`, `with-open` и так далее:

```
(let [a 42
      b "abc"]
  ...)

(if-let [x (test)]
  then
  else)

(with-open [in (input-stream ...)
            out (output-stream ...)]
  ...)

(for [x (range 10)
      y (range x)]
  [x y])
```

`for` — весьма интересный пример, потому что значения инициализирующих выражений не сохраняются в локальных именах: `x` будет хранить не значение выражения `(range 10)`, а последовательно получать все значения `(range 10)`. То есть, вы всегда должны помнить, что *инициализаторы необязательно должны быть значениями*.

Не мудрите при определении переменных. В Clojure имеется множество макросов, определяющих переменные, каждый из кото-

[1] Другие устраняются функциями.

рых опирается на форму `def` или ее производные. Если вы собираетесь написать такой макрос, имейте в виду несколько основных моментов, чтобы семантика определения переменных в вашем макросе соответствовала ожиданиям пользователя:

Макрос, определяющий переменную, должен иметь имя, начинающееся с `def`

Это поставит ваш макрос в один ряд с другими макросами, определяющими переменные, такими как `defn`, `defn-`, `defmacro` и так далее. Префикс `def` подскажет вашему пользователю, что макрос вводит новую переменную, и что он должен использоваться в качестве формы верхнего уровня.

Принимайте имя переменной в первом аргументе

Чтобы не задаваться вопросом о порядке следования аргументов, можно неявно генерировать имена переменных, но такой подход требует от пользователей понимания особенностей реализации вашего макроса, что решительно противоречит их представлениям об инструментах абстракции.

Определяйте не более одной переменной в каждой форме макроса

В дополнение к замечаниям, касающимся обычно не самого лучшего решения определять переменные с автоматически генерируемыми или подразумеваемыми именами: не определяйте более одной переменной в макросе. Так же как вы посчитали бы странным обнаружить возможность формы `def` или `defn` определять несколько переменных, ваши пользователи тоже посчитали бы странным, если бы ваш макрос определял несколько переменных. Единственное исключение из этого правила – отдельные (приватные!) переменные, необходимые макросу, которые пользователь не может использовать и вообще не обязан подозревать об их существовании.

Макросы не должны проявлять сложные формы поведения. В идеале макросы должны быть лишь тонкими обертками вокруг существующих функций (или других макросов) или легко воспроизводиться с их помощью. Это положение возвращает нас к дискуссии в разделе «Когда следует использовать макросы» выше.

Существует множество примеров, противоречащих этому правилу, включая макрос `for`, когда макросы развертываются в весьма

сложный программный код. Однако макрос `for` не делает ничего такого, что нельзя было бы повторить с помощью комбинации `map`, `filter`, `mapcat`, `fn` и `let`. Сложность кода, генерируемого этим макросом, обусловлена лишь необходимостью оптимизации и синтаксическими требованиями: он не скрывает и не заключает в себе какую-то особенную функциональность.

Большую часть своей работы макросы должны делегировать функциям и выполнять только те операции, реализовать которые в виде функций было бы сложнее: управление вычислением.

Неявные аргументы: &env и &form

Выше, в разделе «Предоставление пользователю права выбора имен», упоминалось, что сам макрос `defmacro` является одним из редких в Clojure анафоричных макросов. `defmacro` создает две неявные локальные привязки: `&env` и `&form`.

&env

`&env` содержит ассоциативный массив, ключами которого являются имена[1] всех текущих локальных привязок (значения, соответствующие ключам, в этом ассоциативном массиве не указываются). Он может пригодиться при отладке:

```
(defmacro spy-env []
  (let [ks (keys &env)]
    `(prn (zipmap '~ks [~@ks]))))

(let [x 1 y 2]
  (spy-env)
  (+ x y))
; {x 1, y 2}
;= 3
```

`&env` может также пригодиться для безопасной оптимизации выражений на этапе компиляции. Ниже приводится очень сырая версия макроса, вычисляющего переданное ему выражение на этапе компиляции, если оно не использует локальные привязки, объявленные в контексте, где используется макрос:

---

[1] Обратите внимание, что на метаданные в ключах `&env` нельзя полагаться, в особенности в присутствии локальных псевдонимов.

```
(defmacro simplify
  [expr]
  (let [locals (set (keys &env))]
    (if (some locals (flatten expr))          ❶
      expr                                    ❷
      (do
        (println "Precomputing: " expr)
        (list `quote (eval expr))))))         ❸
```

❶ Здесь выполняется весьма неоптимальный поиск ссылок на локальные привязки в теле кода, переданного макросу.
❷ При обнаружении таких ссылок, код возвращается в неизменном виде.
❸ Иначе макрос вычисляет выражение *на этапе компиляции* и возвращает значение, которое будет подставлено на место выражения (после вывода информационного сообщения, которое поможет нам оставаться в курсе происходящего).

Этот макрос позволяет оптимизировать выражения, которые по нашему мнению могут быть исключены из выполняемого программного кода, при этом неоптимизированные версии выражений остаются в исходном коде без изменений (часто это намного лучше, чем вручную вычислять выражения и подставлять на их место «магические» константы):

```
(defn f
  [a b c]
  (+ a b c (simplify (apply + (range 5e7)))))
; Precomputing:  (apply + (range 5e7))
;= #'user/f
(f 1 2 3)              ;; вернет значение немедленно
;= 1249999975000006
(defn f'
  [a b c]
  (simplify (apply + a b c (range 5e7))))
;= #'user/f'
(f' 1 2 3)             ;; вычисления займут примерно 2.5 сек.
;= 1249999975000006
```

Так как выражение `apply` в функции `f` не содержит ссылок на локальные привязки в ней, макрос `simplify` преобразует его в постоянное значение. Эта оптимизация принимает форму значения, вычисленного макросом `simplify` на этапе компиляции, поэтому на этапе выполнения функция `f` не тратит много времени (около 2.5 сек.) на

суммирование чисел в указанном диапазоне. Выражение, передаваемое макросу `simplify` в функции `f'`, напротив, *зависит* от локальных значений, о чем свидетельствует `&env`, поэтому `simplify` не выполняет оптимизацию, оставляя выражение, которое целиком вычисляется на этапе выполнения.

Тестирование и отладка использования &env. Макросы, использующие `&env`, могут оказаться весьма сложными в отладке. У нас на выбор есть две возможности: либо написать макрос, задействовав прием журналирования или другой способ, позволяющий наблюдать за его действиями (например, добавить вызовы функции `println`, как это было сделано в макросе `simplify`), либо воспользоваться некоторыми особенностями реализации макросов, чтобы протестировать их по отдельности.

В настоящее время макросы реализуются как функции, принимающие два неявных аргумента, предшествующих остальным аргументам, перечисленным в сигнатуре, значения `&form` и `&env`, когда они вызываются компилятором Clojure. Однако, как было показано в разделе «Когда следует использовать макросы», выше, Clojure не позволяет использовать макросы в виде функций, чтобы исключить вероятность появления целого класса ошибок, связанных с этим.

Мы можем обойти это ограничение, дя чего достаточно получить переменную с макросом, извлечь из нее функцию реализации и использовать ее непосредственно – практически так же, как это делает компилятор:

```
(@#'simplify nil {} '(inc 1))          ❶
; Precomputing:  (inc 1)
;= (quote 2)
(@#'simplify nil {'x nil} '(inc x))
;= (inc x)
```

❶ Последовательность знаков перед именем `simplify` выглядит как ругательство – это синтаксический сахар, позволяющий разыменовывать именованные переменные, эквивалентный выражению `(deref (var simplify))`, который может пригодиться для получения значений приватных переменных – и должна служить предупреждением. Данный прием можно использовать для тестирования макросов только в крайнем случае, так как он основан на особенностях текущей реализации Clojure.

Здесь мы вызываем функцию, реализующую макрос, передавая ей два дополнительных аргумента перед тестовым выражением:

аргумент `nil`, соответствующий параметру `&form` (который, как мы знаем, не используется макросом), и ассоциативный массив для параметра `&env`, содержащий или не содержащий ключ гипотетической локальной привязки. Это позволяет увидеть код, который будет произведен макросом `simplify` для двух выражений. Обратите внимание, что в данном случае нельзя протестировать макрос с помощью функции `macroexpand`, потому что они не дают возможности передать фиктивный ассоциативный массив `&env`, как это позволяет функция реализации макроса[1].

&form

`&form` хранит текущую форму вызова макроса целиком, то есть, список, содержащий символ с именем макроса (в том виде, в каком он указан в исходном коде, включая псевдоним или альтернативное имя) и аргументы. Это – самая настоящая форма, прочитанная механизмом чтения[2]. Это означает, что `&form` содержит все метаданные, указанные пользователем, такие как определения типов, и добавленные механизмом чтения, такие как номер строки, где используется макрос.

Рассмотрим два интересных приема использования `&form`: вывод в макросах более информативных сообщений об ошибках времени компиляции и проверка сохранения макросом определений типов, сделанных пользователем.

Вывод сообщений об ошибках в макросах

Одной из ключевых областей применения этой информации является вывод точных и подробных сообщений об ошибках. Например, рассмотрим макрос, принимающий несколько векторов триад[3] – он может вычислять соответствующую онтологию (ontology), предва-

---

[1] В разделе «Тестирование контекстных макросов», ниже, демонстрируется альтернативная версия функции, выполняющей развертывание макроса, которая поддерживает такую возможность без использования операторов разыменования переменной.

[2] Или полученная в результате развертывания внешнего макроса.

[1] В семантике веб-технологий, таких как RDF, термином *триады* обозначаются выражения вида субъект-предикат-объект. Конкретное представление и семантика триад различаются между реализациями, но в упрощенном виде вектор триады можно было бы представить, как: `[«Boston» :capital-of «Massachusetts»]`.

рительно вычисляя различные отношения и свойства этой онтологии, чтобы этого не приходилось делать во время выполнения:

```
(defmacro ontology
  [& triples]
  (every? #(or (== 3 (count %))
               (throw (IllegalArgumentException.
                 "All triples provided as arguments must have 3 elements")))
          triples)
  ;; здесь конструируется и выводится созданная онтология...
  )
```

Исключение будет сгенерировано, только когда один из векторов в аргументе `ontology` содержит не три элемента. Однако сопутствующее сообщение будет весьма далеко от идеала:

```
(ontology ["Boston" :capital-of])                                    ❶
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   All triples provided as arguments must have 3 elements>
(pst)
;= IllegalArgumentException All triples provided as arguments must have 3
;=   elements user/ontology (NO_SOURCE_FILE:3)                       ❷
```

❶ Неполная триада не вызовет исключение, но...

❷ ...номер строки в сообщении об ошибке будет неправильным с точки зрения пользователя. Здесь 3 – это номер строки относительно начала макроса, а не относительно начала файла.

Номер строки в коротком сеансе работы с REPL не играет большой роли, но в настоящем проекте с объемными файлами из-за такой ошибки можно впустую потратить минуты и часы на отладку макроса. Исправить эту проблему можно с помощью списка `&form` и его метаданных:

```
(defmacro ontology
  [& triples]
  (every? #(or (== 3 (count %))
               (throw (IllegalArgumentException.
                        (format
                          "`%s` provided to `%s` on line %s has < 3 elements"
                          %                                          ❶
                          (first &form)                              ❷
                          (-> &form meta :line)))))                  ❸
```

```
          triples)
  ;; ...
  )
```

❶ Мы включили в сообщение вектор, вызвавший ошибку. Это никак не связано с `&form`, но в любом случае лишним не будет.
❷ Первым в списке `&form` всегда следует имя макроса, указанное пользователем. Практическое значение его использования будет показано чуть ниже.
❸ Здесь мы извлекаем номер строки из метаданных, подготовленных механизмом чтения для `&form`. Данный номер строки точно определяет место, где *пользовательский код* использует макрос.

После этих изменений макрос выводит намного больше полезной информации в случае ошибки:

```
(ontology ["Boston" :capital-of])
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=    `["Boston" :capital-of]` provided to `ontology` on line 1 has < 3
;=    elements>
```

Отлично, теперь сообщение об ошибке содержит точный номер строки. Но зачем возиться с выводом имени, которое было использовано для идентификации макроса? Дело в том, что существует возможность переименовывать символы, идентифицирующие переменные, «импортированные» из другого пространства имен, с помощью функции `refer` и ее производных, что бывает очень удобно, когда требуется использовать функции и макросы из других пространств имен, имеющих одинаковые или похожие имена[1]. Поэтому, перейдя в другое пространство имен, мы можем использовать наш макрос `ontology` под другим именем:

```
(ns com.clojurebook.macros)
;= nil
(refer 'user :rename '{ontology triples})
;= nil
```

Теперь в пространстве имен `com.clojurebook.macros` макрос `ontology` будет доступен под именем `triples`. Благодаря нашим исправлени-

[1] Функция `refer` описывается в разделе «refer» (глава 8) и используется ниже в этой главе.

ям, макрос `ontology` будет генерировать текст сообщения об ошибке, точно отражающее имя макроса, *указанное в месте фактического его использования*:

```
(triples ["Boston" :capital-of])
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=    `["Boston" :capital-of]` provided to `triples` on line 1 has < 3
;=    elements>
```

Без использования выражения `(first &form)` при конструировании сообщения об ошибке, в первый момент возникло бы чувство некоторой растерянности, прежде чем несчастный пользователь макроса вспомнил бы, что в текущем пространстве имен он переименовал макрос `ontology` в `triples`.

Сохранение определений типов, сделанных пользователем

Большинство макросов не используют метаданные, присоединяемые к формам в их контексте, включая определения типов[1]. Например, в следующем фрагменте макрос `or` не включает определение типа `^String` в код, который он производит, что приводит к выводу предупреждения механизма рефлексии:

```
(set! *warn-on-reflection* true)
;= true
(defn first-char-of-either
  [a b]
  (.substring ^String (or a b) 0 1))
; Reflection warning, NO_SOURCE_PATH:2 - call to substring can't be
; resolved.
;= #'user/first-char-of-either
```

> **Примечание.** Такие ситуации редко встречаются на практике, потому что определения типов обычно помещаются выше в исходном коде, благодаря чему тип формы макроса определяется с помощью механизма вывода типов:
>
> ```
> (defn first-char-of-either
>   [^String a ^String b]
>   (.substring (or a b) 0 1))
> ;= #'user/first-char-of-either
> ```

[1] Подробнее об определениях типов рассказывается в разделе «Указание типов для производительности», в главе 9.

Мы можем убедиться, что метаданные выражения `or` действительно теряются. Ниже показано выражение с метаданными:

```
(binding [*print-meta* true]
  (prn '^String (or a b)))
; ^{:tag String, :line 1} (or a b)
```

Но если применить `macroexpand` к тому же выражению, метаданные исчезнут:

```
(binding [*print-meta* true]
  (prn (macroexpand '^String (or a b))))
; (let* [or__3548__auto__ a]
;   (if or__3548__auto__ or__3548__auto__ (clojure.core/or b)))
```

Однако нет никаких причин, по которым нельзя было бы сохранить определение типа в выражении `or` – для этого достаточно задействовать `&form` в определении макроса. Для начала посмотрим, как выглядит реализация макроса `or` в `clojure.core`:

```
(defmacro or
  ([] nil)
  ([x] x)
  ([x & next]
    `(let [or# ~x]
       (if or# or# (or ~@next)))))
```

Нам необходимо проверить, содержит ли `&form` метаданные, соответствующие типу, указанному пользователем, и добавить их в выражение, возвращаемое из `or`. В большинстве случаев, чтобы добавить в возвращаемое выражение метаданные из `&form`, достаточно просто заключить тело макроса в вызов `with-meta`, но здесь мы не можем сделать этого[1]. Поэтому мы добавим определение типа к элементу, которому оно принадлежит: к символу:

```
(defmacro OR
  ([] nil)
```

[1] Из-за особенностей реализации: специальным формам (таким как `let` – самая внешняя форма в выражении, возвращаемом макросом `or`) нельзя передать определение типа. Поэтому нам необходимо ввести локальную привязку, чтобы добавить определение типа к ней.

```
  ([x]
    (let [result (with-meta (gensym "res") (meta &form))]
      `(let [~result ~x]
         ~result)))
  ([x & next]
    (let [result (with-meta (gensym "res") (meta &form))]
      `(let [or# ~x
             ~result (if or# or# (OR ~@next))]
         ~result))))
```

Чтобы убедиться, что определение типа сохранилось, можно еще раз воспользоваться функцией `macroexpand` (обратите внимание, что здесь используется наша версия макроса `or`, макрос `OR`):

```
(binding [*print-meta* true]
  (prn (macroexpand '^String (OR a b))))
; (let* [or__1176__auto__ a
;        ^{:tag String, :line 2}
;        res1186 (if or__1176__auto__ or__1176__auto__ (user/or b))]
;   ^{:tag String, :line 2} res1186)
```

Теперь метаданные, определяемые пользователем, не теряются, что в данном случае приводит к исчезновению предупреждения:

```
(defn first-char-of-any
  [a b]
  (.substring ^String (OR a b) 0 1))
;= #'user/first-char-of-any
```

Шаблон, представленный в определении макроса `OR` выше можно разбить на функции, которые затем использовать в других макросах:

```
(defn preserve-metadata
  "Обеспечивает передачу в тело `expr` метаданных из `&form`."
  [&form expr]
  (let [res (with-meta (gensym "res") (meta &form))]
    `(let [~res ~expr]
       ~res)))

(defmacro OR
  "Действует так же, как `clojure.core/or`, но сохраняет пользовательские
   метаданные (такие как определения типов)."
  ([] nil)
```

```
  ([x] (preserve-metadata &form x))
  ([x & next]
    (preserve-metadata &form `(let [or# ~x]
                                (if or# or# (or ~@next))))))
```

Создание версии макроса `defmacro`, которая всегда использовала бы `preserve-metadata` для поддержки пользовательских метаданных в макросах, мы оставим читателям, как самостоятельное упражнение.

Тестирование контекстных макросов

Как было показано выше, макросы, использующие `&form` и `&env`, тяжело поддаются тестированию. Однако, опираясь на наши знания об этих аргументах и о том, как реализуются макросы, мы можем на скорую руку написать свою версию функции `macroexpand-1`, позволяющую передавать фиктивный аргумент `&env` для нужд тестирования и отладки:

```
(defn macroexpand1-env [env form]
  (if-let [[x & xs] (and (seq? form) (seq form))]
    (if-let [v (and (symbol? x) (resolve x))]
      (if (-> v meta :macro)
        (apply @v form env xs)
        form)
      form)
    form))
```

Функцию `macroexpand1-env` можно использовать для проверки поведения контекстного макроса `simplify` в различных «окружениях»:

```
(macroexpand1-env '{} '(simplify (range 10)))
; Precomputing:  (range 10)
;= (quote (0 1 2 3 4 5 6 7 8 9))
(macroexpand1-env '{range nil} '(simplify (range 10)))
;= (range 10)
```

Аналогично можно проверить обработку метаданных `&form` в макросе, добавив эти метаданные к коду, который передается функции `macroexpand1-env`. Например, допустим, что нам необходимо изменить исправлявшийся выше макрос `spy` и добавить в него вывод номера строки, где используется макрос, извлекаемого из метаданных в `&form`:

```
(defmacro spy [expr]
  `(let [value# ~expr]
     (println (str "line #" ~(-> &form meta :line) ",")
              '~expr value#)
     value#))
;= #'user/spy
(let [a 1
      a (spy (inc a))
      a (spy (inc a))]
  a)
; line #2, (inc a) 2
; line #3, (inc a) 3
;= 3
```

Чтобы проверить вывод номера строки, не запуская код, который возвращает макрос, можно воспользоваться функцией `macroexpand1-env`:

```
(macroexpand1-env {} (with-meta '(spy (+ 1 1)) {:line 42}))     ❶
;= (clojure.core/let [value__602__auto__ (+ 1 1)]
;=   (clojure.core/println
;=     (clojure.core/str "line #" 42 ",")                       ❷
;=     (quote (+ 1 1)) value__602__auto__)
;=   value__602__auto__)
```

❶ Мы заменили метаданные `:line` формы, передаваемой функции `macroexpand1-env` на некоторое необычное значение, чтобы сделать его заметнее...

❷ ...и убедились, что наш макрос действительно использует номер строки из метаданных в `&form`.

Тело функции `macroexpand1-env` в действительности является отличным кандидатом для преобразования его в макрос: вложенные выражения `if-let` и повторяющиеся значения `form` смотрятся не особо красиво. Было бы лучше, если бы можно было превратить все эти `if-let` в единственную форму, например так:

```
(defn macroexpand1-env [env form]
  (if-all-let [[x & xs] (and (seq? form) (seq form))
               v (and (symbol? x) (resolve x))
               _ (-> v meta :macro)]
    (apply @v form env xs)
    form))
```

Было бы желательно, чтобы макрос `if-all-let` вычислял форму `then`, только если ни одно из связанных выражений не возвращает `false` или `nil`. Ниже приводится определение такого макроса:

```
(defmacro if-all-let [bindings then else]
  (reduce (fn [subform binding]
            `(if-let [~@binding] ~subform ~else))
    then (reverse (partition 2 bindings))))
```

Он полностью удовлетворяет нашей второй реализации `macroexpand1-env` и может пригодиться везде, где потребуется использовать вложенные формы `if-let`.

Подробности: -> и ->>

Чтобы понять, как действовать при решении настоящих проблем, разработаем альтернативную реализацию макроса, часто используемого в Clojure: `->`, который близко напоминает родственный ему макрос `->>`. Эти макросы, часто называемые *потоковыми макросами* (threading macros), определены в `clojure.core` (существует также множество их производных с похожей семантикой в различных сторонних библиотеках) и особенно полезны для упорядочения цепочек из вызовов функций или вызовов методов, при взаимодействии с Java.

Нам хотелось бы иметь возможность переписать, например такой неуклюжий код:

```
(prn (conj (reverse [1 2 3]) 4))
```

в более удобочитаемом виде:

```
(thread [1 2 3] reverse (conj 4) prn)
```

Чтобы не читать код в направлении изнутри наружу (что может оказаться довольно сложным занятием для глубоко вложенных вызовов), а расположить вызовы последовательно, слева направо, как серию следующих друг за другом операций: «Начать с вектора [1 2 3], переставить элементы в обратном порядке, добавить в него 4, и затем вывести».

Не трудно представить макрос, реализующий такое преобразование. Принимая последовательность форм, мы могли бы вставить первую форму во вторую, на место второго элемента, затем вставить

получившуюся форму в третью, так же на место второго элемента, и так далее.

Кроме того, если какая-то форма, следующая за первой, не является списком, ее можно считать списком с единственным элементом. Это позволит избавиться от скобок в функциях с единственным аргументом, таких, как показано ниже:

```
(-> foo (bar) (baz))
```

и записывать их, как:

```
(-> foo bar baz)
```

Для начала напишем простую вспомогательную функцию, преобразующую данную форму в последовательность.

```
(defn ensure-seq [x]
  (if (seq? x) x (list x)))

(ensure-seq 'x)
;= (x)
(ensure-seq '(x))
;= (x)
```

Теперь, нам нужна функция, принимающая две формы, x и ys, будет вставлять x в ys на место второго элемента, преобразуя ys в последовательность.

```
(defn insert-second
  "Вставляет x на место второго элемента в последовательность y."
  [x ys]
  (let [ys (ensure-seq ys)]
    (concat (list (first ys) x)
            (rest ys))))
```

Это обычная функция, но не следует забывать, что она будет вызываться из макроса. Поэтому значениями параметров x и ys будут структуры данных, представляющие исходный программный код.

Эту функцию можно записать более кратко, воспользовавшись синтаксисом синтаксического маскирования и размаскирования. Не забывайте, что эти механизмы можно использовать за пределами тела макроса:

```
(defn insert-second
  "Вставляет x на место второго элемента в последовательность y."
  [x ys]
  (let [ys (ensure-seq ys)]
    `(~(first ys) ~x ~@(rest ys))))
```

То же самое можно реализовать еще более кратким способом, воспользовавшись функцией `list*`, рассматривавшейся в разделе «Создание последовательностей», в главе 3, и часто используемой в реализациях макросов и их вспомогательных функций:

```
(defn insert-second
  "Вставляет x на место второго элемента в последовательность y."
  [x ys]
  (let [ys (ensure-seq ys)]
    (list* (first ys) x (rest ys))))
```

Теперь напишем макрос с именем `thread`. Форма `(thread x)` должна возвращать просто `x`. Форма `(thread x (a b))` должна возвращать `(a x b)`, задействовав наши вспомогательные функции. Форма `(thread x (a b) (c d))` может выполнить преобразование первых двух элементов и затем рекурсивно применить `thread` к результату.

```
(defmacro thread
  "Вставляет x в последующие формы."
  ([x] x)
  ([x form] (insert-second x form))
  ([x form & more] `(thread (thread ~x ~form) ~@more)))
```

Похоже он работает так же, как стандартный макрос `->`[1]:

```
(thread [1 2 3] (conj 4) reverse println)
;= (4 3 2 1)
(-> [1 2 3] (conj 4) reverse println)
;= (4 3 2 1)
```

Единственное, о чем мы пока не подумали, – возможно ли реализовать этот макрос в виде обычной функции, чтобы вообще из-

[1] Повторная реализация макроса `->>` должна получиться еще более простой, поэтому оставляем ее в качестве самостоятельного упражнения. Подсказка: вместо `insert-second` вам потребуется функция `insert-last`.

бежать использования макросов. Как оказывается, это возможно, внеся небольшие изменения:

```
(defn thread-fns
  ([x] x)
  ([x form] (form x))
  ([x form & more] (apply thread-fns (form x) more)))

(thread-fns [1 2 3] reverse #(conj % 4) prn)
;= (4 3 2 1)
```

В данном случае, необходимость обертывать некоторые вызовы функций конструкцией `#()` делает программный код немного длиннее и сложнее для чтения, а также перегружает наш код посторонними вызовами функций. Кроме того, версия в виде функции не будет работать с вызовами Java-методов, тогда как версия в виде макроса будет, потому что она просто манипулирует списками и символами.

```
(thread [1 2 3] .toString (.split " ") seq)
;= ("[1" "2" "3]")

(thread-fns [1 2 3] .toString #(.split % " ") seq)
;= #<CompilerException java.lang.RuntimeException:
;=   Unable to resolve symbol: .toString in this context,
;=   compiling:(NO_SOURCE_PATH:1)>

;; Код начинает выглядеть все более запутанным...
(thread-fns [1 2 3] #(.toString %) #(.split % " ") seq)
;= ("[1" "2" "3]")
```

Версия в виде макроса определенно выглядит предпочтительнее, чем версия в виде функции. В действительности потоковые макросы в Clojure часто используются для придания большей ясности цепочкам из вызовов функций, применяющим последовательность преобразований к единственному значению или коллекции значений.

Настоящая реализация макроса `->` в действительности ненамного сложнее чем наш макрос `thread`. В `clojure.core` входит еще несколько потоковых макросов, включая:

- `..`

  Действует так же, как `->`, но применяется только к вызовам, взаимодействующим с Java (и к вызовам статических Java-

методов, не поддерживаемых макросом ->). Макрос .. появился до макроса -> и в настоящее время редко используется на практике, однако его все еще можно встретить в природе.

->>

Цепочка вызовов формируется путем добавления каждой предшествующей формы в последующую, на место *последнего* ее элемента, а не второго. Обычно используется для преобразования последовательностей с помощью последовательности функций. Например:

```
(->> (range 10) (map inc) (reduce +))
;= 55
```

Надеюсь, это послужит убедительным примером широких возможностей и практической ценности макросов.

В заключение

Как и в любых других диалектах Lisp, макросы обеспечивают немалую долю выразительности языка Clojure. История развития макросов прошла довольно длинный путь в направлении устранения уродливости и абстрагирования распространенных шаблонов в программном коде. Однако это не означает, что они являются самыми основными конструкциями, к которым следует стремиться в первую очередь: язык Clojure позволяет уйти очень далеко вообще без создания макросов.

Функциональное программирование и моделирование данных уже обладают огромной выразительностью и позволяют абстрагировать шаблоны, наиболее часто встречающиеся в программном коде. Макросы – это последний шаг, упрощающий шаблоны управления потоком выполнения и добавляющий синтаксический сахар, уменьшающий или устраняющий неуклюжий код.

Глава 6. Типы данных и протоколы

При программировании на других языках многие из нас не раз сталкивались с такой ситуацией: вы пишете красивые интерфейсы в соответствии с канонами проектирования и вдруг обнаруживаете, что вынуждены иметь дело с объектом, предоставляемым другим модулем, совершенно неподконтрольным вам. Мало надежды, что разработчики этого модуля добавят поддержку для вашего интерфейса (по техническим, политическим или юридическим причинам). И в один момент вы начинаете тонуть в адаптерах и прокси-объектах.

Наиболее удачливые из нас работают с языками, достаточно динамичными, чтобы позволить *латание по-обезьяньи* (monkey-patching), где классы открыты и дают возможность в любой момент внедрить дополнительные методы, чтобы связать два интерфейса. К счастью, этот термин звучит достаточно уничижительно, чтобы заставить вас вздрогнуть и дважды подумать, прежде чем использовать данный прием, поскольку подобное хирургическое вмешательство сопряжено с массой сложностей и скрытых ловушек.

Основная проблема, стоящая за всем этим, получила название *проблемы выразительности* (expression problem):

> Проблема Выразительности – это новое название старой проблемы. Цель состоит в том, чтобы на основе типа данных для случаев иметь возможность добавлять новые случаи и новые функции для работы с ним без перекомпиляции существующего кода, сохранив при этом статическую безопасность типа.
>
> – Филип Вадлер (Philip Wadler),
> http://www.daimi.au.dk/~madst/tool/papers/expression.txt

Некоторые утверждают, что проблема выразительности в динамических языках остается все той же проблемой выразительности, как ее первоначально обозначил Филип Вадлер. В этой книге мы не собираемся выяснять, сколько ангелов могут одновременно танцевать на кончике иглы, поэтому нам достаточно будет заняться собствен-

ной *проблемой динамической выразительности*. Перефразировав более объектно-ориентированное выражение:

> Проблема *динамической* выразительности – это новое название старой проблемы. Цель состоит в том, чтобы на основе интерфейсов и типа иметь возможность создавать новые типы, реализующие имеющиеся интерфейсы, и предоставлять реализацию новых интерфейсов для имеющегося типа без перекомпиляции существующего кода.

Первая цель – создание новых типов, реализующих старые интерфейсы, – достигается достаточно просто: эта возможность поддерживается всеми объектно-ориентированными языками. Вторая – реализация новых интерфейсов для старых типов – является наиболее интересной частью проблемы, и лишь немногие языки пытались решить ее. Теперь мы посмотрим, как она решается в языке Clojure.

Протоколы

Интерфейсы в языке Clojure называются *протоколами* (protocols)[1]. Термин *интерфейс* (interface) лучше оставить для обозначения интерфейсов Java.

Протокол состоит из одного или более методов, где каждый метод может иметь разное количество параметров. Все методы имеют хотя бы один параметр, соответствующий привилегированной ссылке `this` в Java и `self` – в Ruby и Python.

Первый аргумент любого метода в протоколе называется привилегированным, потому что в зависимости от него, а точнее от его типа, при вызове будет выбираться конкретная реализация метода. Как следствие, выбор реализации в протоколах зависит только от типа единственного значения. Это – весьма ограниченная форма полиморфизма, но она была выбрана для протоколов по вполне прагматичным причинам, а именно:

- эта форма реализована в большинстве виртуальных машин (таких как JVM, CLR и JavaScript) и, что более важно, оптимизирована;
- несмотря на ограниченность, эта форма полиморфизма отвечает широкому кругу потребностей.

Если вам потребуется по-настоящему неограниченный или множественный полиморфизм, для этого в Clojure имеются мультиметоды (multimethods), описываемые в главе 7.

---

[1] Этот термин должен быть знаком программистам со знанием Smalltalk или Objective C.

Ниже показано, как выглядит определение протокола:

```
(defprotocol ProtocolName
  "documentation"
  (a-method [this arg1 arg2] "method docstring")
  (another-method [x] [x arg] "docstring"))          ❶
```

❶ Привилегированный аргумент передается явно, поэтому ему можно дать любое имя по своему выбору: `this`, `self`, `_`, и так далее.

В отличие от большинства других имен в Clojure, имена протоколов и типов записываются символами `ПеременногоРегистра`, потому что компилируются в «родные» интерфейсы и классы JVM. Это позволяет легко отличать протоколы и типы от других сущностей в Clojure, и допускает идиоматическое использование их в других языках, основанных на JVM.

С точки зрения пользователя, *методы протоколов являются функциями*: для обращения к ним не требуется использовать какой-то специальный синтаксис, из них можно извлекать строки документирования с помощью `doc`, их можно передавать функциям высшего порядка, и так далее. Однако протоколы должны проектироваться с позиции разработчика, а не пользователя. Хороший протокол должен содержать небольшое количество методов, с неперекрывающейся функциональностью. Хороший протокол *прост в реализации*. Нет ничего плохого, если протокол определяет единственный метод.

> **Внимание.** Несмотря на то, что методы протокола являются функциями, вы не сможете использовать механизм деструктуризации или извлекать остальные аргументы в спецификации метода внутри формы `def protocol`. Например, опираясь на знания, полученные в разделе «Деструктуризация аргументов функций», в главе 1, можно было бы подумать, что следующее объявление определяет протокол с единственным методом, принимающим два позиционных аргумента и некоторое количество дополнительных аргументов, собранных в последовательность `more`:
>
> ```
> (defprotocol AProtocol
>   (methodName [this x y & more]))
> ```
>
> Однако, из-за того, что протоколы генерируют интерфейсы JVM, которые поддерживают не все способы организации аргументов функций, возможные в языке Clojure, `methodName` фактически является методом, принимающим четыре и только четыре аргумента.

Вспомогательные или служебные функции не обязательно должны быть частью протокола. Они должны лишь опираться на методы протокола.

При необходимости функции, доступные пользователю, и методы протокола могут даже принадлежать разным пространствам имен, чтобы более явно обозначить, что является частью общедоступного API, а что – частью API, которым пользуются разработчики[1]. Пример такого разделения можно найти в самом языке Clojure: пространство имен `clojure.core.protocols` включает протокол `InternalReduce`:

```
(defprotocol InternalReduce
  "Протокол конкретных типов последовательностей, реализующих собственный
   механизм свертки, более быстрый, чем рекурсия first/next. Вызывается
   функцией clojure.core/reduce."
  (internal-reduce [seq f start]))
```

Как пользователи языка Clojure и его операций с последовательностями, мы можем не пользоваться этим протоколом и даже не знать о его существовании. Однако, если нам потребуется заняться разработкой собственных структур данных и обеспечить оптимизированную реализацию свертки, опираясь на некоторые специфические факторы, реализация этого протокола будет представлять для нас определенный интерес.

В процессе знакомства с протоколами и типами данных, мы будем использовать следующий протокол `Matrix`, определяющий операции доступа к элементам двухмерных упорядоченных структур данных (таких как массивы, векторы или их комбинации) и их изменения:

```
(defprotocol Matrix
  "Протокол для работы с 2-мерными структурами данных."
  (lookup [matrix i j])
  (update [matrix i j value])
  (rows [matrix])
  (cols [matrix])
  (dims [matrix]))
```

Расширение существующих типов

Наша первая версия этого протокола вектора векторов будет очень простой. «Вектор векторов» не является типом данных языка

[1] Это в точности соответствует разделению API/SPI (service provider interface – интерфейс службы) во многих Java-библиотеках.

Clojure и также не является типом данных JVM[1], поэтому мы просто расширим протокол векторов в Clojure:

```
(extend-protocol Matrix
  clojure.lang.IPersistentVector
  (lookup [vov i j]
    (get-in vov [i j]))
  (update [vov i j value]
    (assoc-in vov [i j] value))
  (rows [vov]
    (seq vov))
  (cols [vov]
    (apply map vector vov))
  (dims [vov]
    [(count vov) (count (first vov))]))
```

Давайте остановимся и познакомимся поближе с макросом `extend-protocol`. Первый его аргумент – это имя протокола (в данном случае `Matrix`). Далее следует череда символов (соответствующих именам типов, таких как `IPersistentVector` – основной интерфейс векторов в Clojure) и списки (реализации методов для прежде объявленного типа, расширяемого протоколом).

Реализации методов не отличаются от обычных функций. В отличие от Ruby или Java здесь отсутствует неявный параметр `self` или `this`: привилегированный первый аргумент, определяющий выбор той или иной реализации протокола, передается методу явно. В этом отношении функции, реализующие протокол, напоминают методы в языке Python.

Следует отметить, что *протоколы не требуют обязательной реализации всех методов*: Clojure просто будет возбуждать исключение при попытке вызвать нереализованный метод[2].

Макрос `extend-protocol` – не единственный способ распространения протоколов на типы. Имеются также:

- возможность определять встроенные (inline) реализации;
- макрос `extend`;
- макрос `extend-type`.

---

[1] Шаблоны (generics) в Java отсутствуют на уровне байт-кода, они исчезают на этапе компиляции – это называется *затиранием типов* (type erasure) и является побочным эффектом, упрощающим жизнь разработчиков, пользующихся динамическими языками.

[2] Фактический тип исключения зависит от особенностей реализации протокола, поэтому не следует полагаться на него.

О встроенных реализациях и о макросе `extend` будет рассказываться ниже, в разделах «Встроенная реализация» и «Повторное использование реализаций», соответственно.

Макрос `extend-type` является парным для макроса `extend-protocol`: макрос `extend-protocol` позволяет распространить один протокол на несколько типов, а макрос `extend-type` — несколько протоколов на один тип. В них используется одна и та же схема определения, только имена протоколов и типов меняются местами.

Пример 6.1. Сравнение особенностей использования extend-type и extend-protocol

```
(extend-type AType
  AProtocol
  (method-from-AProtocol [this x]
    (;...реализация для типа AType
     ))
  AnotherProtocol
  (method1-from-AnotherProtocol [this x]
    (;...реализация для типа AType
     ))
  (method2-from-AnotherProtocol [this x y]
    (;...реализация для типа AType
     )))

(extend-protocol AProtocol
  AType
  (method1-from-AProtocol [this x]
    (;...реализация для типа AType
     ))
  AnotherType
  (method1-from-AProtocol [this x]
    (;...реализация для типа AnotherType
     ))
  (method2-from-AProtocol [this x y]
    (;...реализация для типа AnotherType
     )))
```

Протокол можно также распространить на значение `nil`, благодаря чему можно распрощаться со многими исключениями обращения к пустому указателю, реализовав соответствующее поведение по умолчанию:

```
(extend-protocol Matrix
  nil
```

```
  (lookup [x i j])
  (update [x i j value])
  (rows [x] [])
  (cols [x] [])
  (dims [x] [0 0]))

(lookup nil 5 5)
;= nil
(dims nil)
;= [0 0]
```

Помощь в использовании протокола и реализации вектора векторов может оказать *фабричная функция* (factory function)[1], создающая пустой вектор векторов определенных размеров.

```
(defn vov
  "Создает вектор из h векторов, каждый из которых содержит w элементов."
  [h w]
  (vec (repeat h (vec (repeat w nil)))))
```

Теперь поэкспериментируем:

```
(def matrix (vov 3 4))
;= #'user/matrix
matrix
;= [[nil nil nil nil]
;=  [nil nil nil nil]
;=  [nil nil nil nil]]
(update matrix 1 2 :x)
;= [[nil nil nil nil]
;=  [nil nil :x nil]
;=  [nil nil nil nil]]
(lookup *1 1 2)
;= :x
(rows (update matrix 1 2 :x))
;= ([nil nil nil nil]
;=  [nil nil :x nil]
```

[1] Фабричные функции в языке Clojure играют роль конструкторов и статических фабричных методов в других языках. Выше вы уже видели некоторые фабричные функции, имеющиеся в Clojure, например, `hash-map`, `sorted-set`, и другие. И, как описывается в разделе «Конструкторы и фабричные функции», ниже, вы так же будете должны определять собственные фабричные функции.

```
;=  [nil nil nil nil])
(cols (update matrix 1 2 :x))
;= ([nil nil nil]
;=  [nil nil nil]
;=  [nil :x nil]
;=  [nil nil nil])
```

Отлично, все работает прекрасно. Мы фактически *распространили* (реализовали) протокол на существующий тип – на векторы. Сделано это было обычным способом, с учетом пространств имен: поскольку методы каждого протокола соответствуют множеству функций в определенном пространстве имен, протоколы не будут конфликтовать друг с другом, даже если будут содержать методы с идентичными именами.

Чтобы показать возможность распространения протокола на любой тип Java, добавим еще одну реализацию протокола `Matrix`, на этот раз для двухмерных массивов вещественных чисел типа `double`, которые в отличие от `IPersistentVector` трудно заподозрить в предоставлении специальной поддержки протоколов.

Пример 6.2. Распространение протокола на массив Java

```
(extend-protocol Matrix
  (Class/forName "[[D")                    ❶
  (lookup [matrix i j]
    (aget matrix i j))
  (update [matrix i j value]               ❷
    (let [clone (aclone matrix)]
      (aset clone i
        (doto (aclone (aget clone i))
          (aset j value)))
      clone))
  (rows [matrix]
    (map vec matrix))
  (cols [matrix]
    (apply map vector matrix))
  (dims [matrix]
    (let [rs (count matrix)]
      (if (zero? rs)
        [0 0]
        [rs (count (aget matrix 0))]))))
```

❶ Вызов `Class/forName` позволяет получить ссылку на класс, соответствующий двухмерному массиву вещественных чисел типа `double`, до-

ступный в Java как `double[][].class`; подробнее об этой форме записи рассказывается в разделе «Классы массивов» (в главе 11).

❷ Массивы *не* являются неизменяемыми структурами данных, но посредством протокола `Matrix` мы отчасти реализовали семантику неизменяемости, обеспечив согласованность с предыдущей реализацией вектора векторов: при каждом изменении создается копия внешнего массива массивов, копии измененных строк, и выполняется подстановка копии с измененным значением в верхнеуровневый возвращаемый массив. Конечно, неограниченная изменчивость может быть желательной для выполнения интенсивных операций с матрицами, поэтому, вероятно, имеет смысл определить несколько отдельных протоколов для таких противоположных семантик.

Эта реализация будет автоматически выбрана протоколом, исходя из типа первого аргумента, передаваемого в каждую функцию; ниже можно видеть, что матрицы на основе массивов действуют в соответствии с нашими ожиданиями:

```
(def matrix (make-array Double/TYPE 2 3))
;= #'user/matrix
(rows matrix)
;= ([0.0 0.0 0.0]
;=  [0.0 0.0 0.0])
(rows (update matrix 1 1 3.4))
;= ([0.0 0.0 0.0]
;=  [0.0 3.4 0.0])
(lookup (update matrix 1 1 3.4) 1 1)
;= 3.4
(cols (update matrix 1 1 3.4))
;= ([0.0 0.0]
;=  [0.0 3.4]
;=  [0.0 0.0])
(dims matrix)
;= [2 3]
```

Теперь должно быть ясно, что несмотря на сходство с интерфейсами Java, протоколы в Clojure обладают более широкими возможностями и не страдают динамической разновидностью *проблемы выразительности*: мы можем привести практически любой существующий тип данных в соответствие с требованиями протоколов, не изменяя эти типы и даже не имея доступа к ним.

До сих пор мы лишь распространяли протоколы на типы, определенные в Java. К счастью Clojure обладает собственными возможностями определения новых типов.

Определение собственных типов

Типы данных в языке Clojure *являются* классами Java, однако определение типов выглядит проще, как показано ниже:

```
(defrecord Point [x y])
```

или

```
(deftype Point [x y])
```

Сначала мы рассмотрим общие черты `deftype` и `defrecord`, а затем углубимся в изучение их различий.

Обе формы определяют новый Java-класс `Point` с двумя общедоступными финальными полями `x` и `y`. Подобно протоколам (и в противоположность другим именам в языке Clojure), имена типов обычно записываются символами с `ПеременнымРегистром`, а не как обычно, символами нижнего регистра с дефисом между словами, потому что они компилируются в Java-классы. Создание нового экземпляра `Point` выполняется простым вызовом конструктора – `(Point. 3 4)` – с аргументами для всех полей, следующими в том же порядке, в каком поля перечислены в определении *типа* или *записи* (record). Так как они являются обычными полями объектов Java:

- чтение и изменение их значений выполняется намного быстрее, чем те же операции, например, с ассоциативными массивами Clojure;
- для обращения к полям экземпляров, объявленным в формах `deftype` или `defrecord`, можно использовать стандартный для Clojure синтаксис доступа к полям[1]:

```
(.x (Point. 3 4))
;= 3
```

По умолчанию каждое поле получает тип `java.lang.Object`, чего вполне достаточно для большинства моделей, и соответствует потребности, если типы некоторых значений в модели могут изменяться. Однако, при необходимости поля можно объявить простыми типами, используя все те же метаданные, что использовались для объявления типов значений, принимаемых и возвращаемых функциями. При желании можно также добавлять подсказки, указываю-

[1] Записи предоставляют более гибкую абстракцию доступа к полям, которая описывается в разделе «Записи – это ассоциативные коллекции» (ниже).

щие на непростые типы полей, как в вызовах методов, однако такие подсказки не изменяют фактические типы полей, которые будет видеть, например, Java-пользователь данного типа.

Таким образом, следующая строка определяет тип записи с полями `x` и `y` простых типов `long`, и с полем `name` типа `Object`, для которого добавлена подсказка, описывающее его как поле типа `String` внутри любых встроенных (inline) реализаций методов, которые могут быть добавлены в тип:

```
(defrecord NamedPoint [^String name ^long x ^long y])
```

Подробнее об использовании подсказок и об объявлениях типов будет рассказываться в разделах «Указание типов для производительности» в главе 9 и «Объявление функций, принимающих и возвращающих значения простых типов» в главе 11, соответственно, однако ни те ни другие не имеют особенно важного значения для понимания того, как действуют макросы `deftype` и `defrecord`.

Иногда бывает полезно знать типы полей, особенно в записях, которые могут поддерживать вспомогательные поля, помимо тех, что указываются в вызовах конструкторов. Перечень объявленных полей называют *базисом* и получить его можно с помощью статического метода класса, определяемого посредством `deftype` или `defrecord`:

```
(NamedPoint/getBasis)
;= [name x y]
```

Каждый символ внутри базиса содержит метаданные, сопровождающие оригинальный вектор полей, включая информацию о типе:

```
(map meta (NamedPoint/getBasis))
;= ({:tag String} {:tag long} {:tag long})
```

Сосредоточимся сначала на записях: они предназначены для использования в качестве модели и представления данных уровня приложения[1], тогда как макрос `deftype` служит для определения типов низкоуровневой инфраструктуры, например, для реализации новой структуры данных.

Различия между этими двумя механизмами целиком заключены в умолчаниях, поддерживаемых записями в терминах взаимодействий Clojure с некоторыми механизмами Java, тогда как `deftype` обеспечи-

[1] Можно сказать, что записи и ассоциативные массивы являются «POJO» (Plain Old Java Object – простой Java-объект) в мире Clojure.

вает возможность оптимизации низкоуровневых операций. Наконец, со временем вы обнаружите, что в большинстве случаев записи используются намного чаще, чем типы, объявленные с помощью `deftype`.

Типы не определены в пространствах имен

Когда с помощью `defrecord` или `deftype` определяется новый тип, получившийся класс помещается в Java-пакет, соответствующий текущему пространству имен, и неявно импортируется в пространство имен, где он определяется, поэтому на него можно сослаться, используя неквалифицированное имя. Однако, когда пространство имен, где определены требуемые типы, импортируется из другого пространства имен, типы не импортируются, потому что являются классами, а не переменными. *Типы необходимо импортировать явно, даже если было выполнено импортирование пространства имен с помощью* `use` *или* `require`.

```
(def x "hello")
;= #'user/hello
(defrecord Point [x y])
;= user.Point                          ❶
(Point. 5 5)
;= #user.Point{:x 5, :y 5}
(ns user2)
(refer 'user)                          ❷
x                                      ❸
;= "hello"
Point                                  ❹
;= CompilerException java.lang.Exception:
;=   Unable to resolve symbol: Point
(import 'user.Point)                   ❺
Point
;= user.Point
```

❶ Объявление типа `Point` в пространстве имен `user` определит новый Java-класс `user.Point`. Неквалифицированное имя типа можно использовать непосредственно, потому что он неявно импортируется в текущее пространство имен.

❷ `refer` напоминает `use`, но не обладает семантикой загрузки, предполагая, что пространство имен уже существует.

❸ `x` теперь *относится* к текущему пространству имен (`user2`).

❹ Однако тип `Point` оказался недоступным.

❺ Тип `Point`, который фактически является классом, необходимо импортировать в текущее пространство имен, чтобы получить возможность ссылаться на него, используя короткое имя.

Подробнее о тонкостях пространств имен в Clojure можно узнать в разделе «Определение и использование пространств имен», в главе 8.

Записи

Типы, определяемые с помощью макроса `defrecord` и часто называемые *типами записей*, являются специализациями типов, определяемых с помощью `deftype`. В число дополнительных особенностей записей входят:

- семантика значения;
- полное соответствие абстракции ассоциативных коллекций;
- поддержка метаданных;
- поддержка механизмом чтения, благодаря чему экземпляры типов записей могут создаваться просто чтением данных;
- дополнительный конструктор для создания записей с метаданными и вспомогательными полями.

> **Внимание.** В Clojure сохраняется частичная реализация «структур» – в виде функций `def-struct`, `create-struct`, `struct-map` и `struct` – которую на данный момент следует считать устаревшей и желательно избегать ее. Структуры зачастую с успехом могут заменить ассоциативные массивы, как описывается в разделе «Ассоциативные массивы как специализированные структуры», в главе 3. В любом случае, записи являются более совершенными структурами данных, чем позволяет получить устаревшая реализация `struct-map`. Об особенностях выбора между этими вариантами мы поговорим в разделе «Когда использовать ассоциативные массивы, а когда записи», ниже.

Семантика значения. Семантика значения предполагает, что записи являются неизменяемыми и, если поля двух записей равны, то и сами записи равны:

```
(defrecord Point [x y])
;= user.Point
(= (Point. 3 4) (Point. 3 4))
;= true
(= 3 3N)
;= true
(= (Point. 3N 4N) (Point. 3 4))
;= true
```

Эта семантика должна быть привычна по работе со структурами данных языка Clojure, дающими те же гарантии. Записи получают эту семантику автоматически за счет методов `Object.equals` и `Object.hashCode`, реализация которых зависит от значений полей записи.

Записи – это ассоциативные коллекции. Записи поддерживают абстракцию ассоциативных коллекций[1], поэтому к записям можно применять те же операции, что и к ассоциативным массивам.

Например, к полям типов и записей можно обращаться, используя формы вида `(.x instance)`, однако к полям записей можно также обращаться как к функциям – в частности, ключевые слова, как описывается в разделе «Упрощенный доступ к коллекциям» в главе 3, являются функциями, которые отыскивают сами себя в ассоциативной коллекции, относительно которой они вызываются. Такие вызовы работают с записями точно так же, как с ассоциативными массивами:

```
(:x (Point. 3 4))          ❶
;= 3
(:z (Point. 3 4) 0)        ❷
;= 0
(map :x [(Point. 3 4)
         (Point. 5 6)
         (Point. 7 8)])
;= (3 5 7)
```

❶ Обратите внимание, что когда ключевое слово используется явно и его литерал находится в позиции функции, компилятор способен оптимизировать такую операцию доступа до уровня простого разыменования поля `(.x instance)`.

❷ Для ключевых слов можно также указывать значения по умолчанию, на случай, если они не определены в записи.

Далее, изменить значение поля можно простым вызовом `assoc`. Кроме того, в вашем распоряжении все остальные функции для работы с ассоциациями и коллекциями, включая `keys`, `get`, `seq`, `conj`, `into` и так далее. И так же как ассоциативные массивы, записи реализуют интерфейс `java.util.Map`, благодаря чему их можно передавать программному коду на языке Java, извлекающему данные из экземпляров `Map`.

При определении записей, в них указывается фиксированное множество полей, однако они позволяют добавлять в них новые слоты, не являющиеся частью первоначального множества[2]:

---

[1] Определяемую интерфейсом `clojure.lang.Associative` и описываемую в разделе «Ассоциативные коллекции», в главе 3.

[2] Эта особенность, когда тип или объект с фиксированным множеством предопределенных полей может «расширяться» и включать другие значения, в других языках и фреймворках иногда называется как *способность к расширению* (expando).

```
(assoc (Point. 3 4) :z 5)                ❶
;= #user.Point{:x 3, :y 4, :z 5}
(let [p (assoc (Point. 3 4) :z 5)]       ❷
  (dissoc p :x))
;= {:y 4, :z 5}
(let [p (assoc (Point. 3 4) :z 5)]       ❸
  (dissoc p :z))
;= #user.Point{:x 3, :y 4}
```

❶ В записи можно добавлять дополнительные слоты.
❷ В результате применения операции `dissoc` к полю, объявленному в определении, возвращается уже не запись, а простой ассоциативный массив.
❸ Однако при удалении дополнительного поля, возвращаемое значение не вырождается в простой ассоциативный массив.

Обратите внимание, что дополнительные слоты хранятся «отдельно» от предопределенных полей, внутри собственной ассоциативной структуры, и обладают соответствующими характеристиками производительности операций поиска и изменения. То есть, они не становятся новыми полями в Java-классе, лежащем в основе записи:

```
(:z (assoc (Point. 3 4) :z 5))
;= 5
(.z (assoc (Point. 3 4) :z 5))
;= #<java.lang.IllegalArgumentException:
;=   No matching field found: z for class user.Point>
```

Поддержка метаданных. Записи, как и любые другие коллекции в языке Clojure, позволяют присваивать им извлекать метаданные с помощью `meta` и `with-meta` (и, как следствие, `vary-meta`) не влияя на семантику значений:

```
(-> (Point. 3 4)
  (with-meta {:foo :bar})
  meta)
;= {:foo :bar}
```

Обзор метаданных и примеры их использования можно найти в разделе «Метаданные», в главе 3.

Читаемое представление. Как можно было заметить, интерактивная оболочка REPL выводит экземпляры записей в определенном

виде, позволяющем отличать их от обычных ассоциативных массивов и произвольных Java-объектов:

```
#user.Point{:x 3, :y 4, :z 5}
```

Это – литерал записи, эквивалентный квадратным скобкам, используемым для обозначения векторов, где символы, начинающиеся с двоеточия, обозначают ключевые слова. Это означает, что вы можете выводить и читать экземпляры записей в текстовом представлении, как любые другие литералы в языке Clojure:

```
(pr-str (assoc (Point. 3 4) :z [:a :b]))
;= "#user.Point{:x 3, :y 4, :z [:a :b]}"
(= (read-string *1)
   (assoc (Point. 3 4) :z [:a :b]))
;= true
```

Это позволяет легко и просто использовать записи для сохранения и извлечения данных (в файлах, в базе данных или где-то еще), как любые другие значения, поддерживаемые механизмом чтения Clojure.

Дополнительный конструктор. В дополнение к конструктору, принимающему предопределенные поля, записи также имею конструктор, поддерживающий некоторые дополнительные особенности, а именно: возможность добавления дополнительных полей и метаданных. Этот второй конструктор принимает два дополнительных аргумента: ассоциативный массив со слотами, определяющими дополнительные поля, сверх тех, что были указаны в определении записи, и ассоциативный массив с метаданными, присоединяемыми к создаваемой записи:

```
(Point. 3 4 {:foo :bar} {:z 5})
;= #user.Point{:x 3, :y 4, :z 5}
(meta *1)
;= {:foo :bar}
```

Это семантический эквивалент (но более эффективный) пошаговому изменению базового экземпляра записи, как показано ниже:

```
(-> (Point. 3 4)
  (with-meta {:foo :bar})
  (assoc :z 5))
```

Конструкторы и фабричные функции

Обычно конструкторы не должны являться частью общедоступного API. Вместо них следует предоставлять доступ к одной или нескольким фабричным функциям, чтобы:

1. Обеспечить максимальное соответствие ожидаемым приемам использования в вызывающем коде, которому не всегда требуется доступ к низкоуровневым конструкторам, сгенерированным моделью, лежащей в основе форм `deftype` и `defrecord`.
2. Иметь возможность применять их на манер функций высшего порядка и легко создавать экземпляры типов и записей из фрагментов «обычных» данных.
3. Чтобы обеспечить неизменность внешнего API, даже когда внутренняя реализация модели претерпевает изменения.

Фабричные функции не столь «хрупкие», как конструкторы при изменении реализации: при изменении списка полей в записи меняется и сигнатура конструктора, кроме того, фабричные функции позволяют добавлять логику, например проверку аргументов или заполнение полей вычисленными значениями и значениями по умолчанию. Типы в языке Clojure не позволяют определять собственные конструкторы, поэтому все, что в других языках можно поместить в тело конструктора, в Clojure следует помещать в фабричные функции.

Оба макроса, `deftype` и `defrecord`, неявно создают одну фабричную функцию вида `->MyType`, принимающую значения полей:

```
(->Point 3 4)
;= #user.Point{:x 3, :y 4}
```

Для записей генерируется также еще одна фабричная функция вида `map->MyType`, которая принимает единственный ассоциативный массив, используемый для заполнения полей нового экземпляра записи:

```
(map->Point {:x 3, :y 4, :z 5})
;= #user.Point{:x 3, :y 4, :z 5}
```

Они обе могут пригодиться для создания экземпляров типов и записей из данных языка Clojure, в особенности в сочетании с функциями высшего порядка:

```
(apply ->Point [5 6])
;= #user.Point{:x 5, :y 6}

(map (partial apply ->Point) [[5 6] [7 8] [9 10]])
;= (#user.Point{:x 5, :y 6}
;=  #user.Point{:x 7, :y 8}
;=  #user.Point{:x 9, :y 10})

(map map->Point [{:x 1 :y 2} {:x 5 :y 6 :z 44}])
;= (#user.Point{:x 1, :y 2}
;=  #user.Point{:x 5, :y 6, :z 44})
```

В случае с записями, фабричная функция на основе ассоциативного массива доступна также как статический метод `create` типа, который может быть очень полезен пользователям библиотек и типов, определяемых в Clojure, из Java:

```
(Point/create {:x 3, :y 4, :z 5})
;= #user.Point{:x 3, :y 4, :z 5}
```

Несмотря на бесспорную полезность фабричных функций, предоставляемых автоматически, иногда возникает необходимость создавать свои собственные функции, чтобы добавить в них дополнительную логику, реализовать заполнение или проверку полей:

```
(defn log-point
  [x]
  {:pre [(pos? x)]}
  (Point. x (Math/log x)))

(log-point -42)
;= #<AssertionError java.lang.AssertionError: Assert failed: (pos? x)>
(log-point Math/E)
;= #user.Point{:x 2.718281828459045, :y 1.0}
```

Часто фабричные функции возникают спонтанно, в процессе разработки, как промежуточный шаг на пути от ассоциативных массивов к записям, когда вы сначала проводите рефакторинг своего кода, вводя фабричные функции для создания ассоциативных массивов, чтобы сделать код чище. Например:

```
(defn point [x y]
  {:x x, :y y})
```

Когда позднее возникает желание перейти на использование записей, вам остается только переписать свою фабричную функцию, чтобы использовать в ней новый тип записи. Часто это продолжать ссылаться на фабричную функцию в клиентском коде, не изменяя его.

Когда использовать ассоциативные массивы, а когда записи

Несмотря на многообразие вариантов применения записей[1], часто предпочтительнее бывает сначала пытаться решить задачу с использованием обычных ассоциативных массивов, а затем переходить к использованию записей, если возникнет такая необходимость.

Ассоциативные массивы позволяют быстро приступить к созданию программного кода и моделированию данных, потому что они не вынуждают заранее определять какие-либо типы или схемы, и тем самым дают большую концептуальную свободу при прототипировании. Однако, как только возникает необходимость в полиморфизме на основе типов (посредством протоколов для записей и типов) или высокоэффективном способе доступа к полям, можно перейти на использование записей: большая часть (если не весь) вашего кода будет продолжать работать, благодаря общим абстракциям для ассоциативных массивов и записей.

Одна из ловушек, ожидающих вас при переходе от ассоциативных массивов к записям, заключается в том, что *записи не являются функциями*. Поэтому выражение `((Point. 3 4) :x)` будет продолжать работать, а выражение `({:x 5 :y 6} :x)` нет. Однако, если следовать рекомендациям из раздела «Упрощенный доступ к коллекциям» в главе 3, касающихся использования ассоциативных массивов в роли функций, случаи, когда ассоциативные используются как функции и когда их можно заменить записями, не будут перекрываться.

Другая ловушка заключается в том, что ассоциативные массивы и записи не могут быть равными, поэтому вам придется уделить особое внимание при смешивании для представления одних и тех же данных.

```
(defrecord Point [x y])
;= user.Point
(= (Point. 3 4) (Point. 3 4))
;= true
```

---

[1] В главе 18 приводится диаграмма, которая при необходимости поможет вам сделать выбор.

```
(= {:x 3 :y 4} (Point. 3 4))
;= false
(= (Point. 3 4) {:x 3 :y 4})    ❶
;= false
```

❶ К счастью Clojure не нарушает симметрию при сравнении.

Типы

`deftype` – самая низкоуровневая в Clojure форма определения типов. В действительности `defrecord` – это всего лишь макрос, построенный на основе `deftype`. Как можно было бы ожидать, многие из «удобств» записей недоступны в типах, определенных с помощью `deftype`. Эта форма в первую очередь предназначена для определения низкоуровневой инфраструктуры типов, необходимой для реализации новых структур данных или ссылочных типов. Ассоциативные массивы и записи, напротив, должны использоваться для хранения данных прикладного уровня.

Эта низкоуровневая форма определения типов предлагает одну важную особенность, без которой *порой* не обойтись на самых низких уровнях приложения или библиотеки: изменяемые поля!

Прежде чем перейти к изменяемым полям, проясним сначала, что обычные (неизменяемые) поля в экземплярах `deftype` доступны только посредством форм взаимодействий:

```
(deftype Point [x y])
;= user.Point
(.x (Point. 3 4))
;= 3
(:x (Point. 3 4))
;= nil
```

Типы, объявленные с помощью `deftype`, не являются ассоциативными[1], поэтому распространенный способ использования ключевых слов в роли функций доступа не может использоваться. В связи с этим необходимо опираться на тот факт, что типы компилируются в Java-классы, в которых все неизменяемые поля определены как общедоступные и финальные (`final`).

---

[1] Если только вы не объявили тип, как реализующий интерфейс `clojure.lang.Associative`; чем-то подобным мы займемся в разделе «Поддержка абстракций коллекций», ниже.

Изменяемые поля могут быть одной из двух разновидностей: изменчивые (volatile) и несинхронизированные (unsynchronized). Чтобы объявить поле изменяемым, его необходимо квалифицировать параметром в метаданных `^:volatile-mutable` или `^:unsynchronized-mutable`, например:

```
(deftype MyType [^:volatile-mutable fld])
```

Параметры `*-mutable` можно сочетать между собой и добавлять в объявления любых полей.

Понятие «изменчивый» (volatile) здесь имеет тот же смысл, что и модификатор полей `volatile` в Java: операции чтения и записи выполняются атомарно[1] и порядок их выполнения должен соответствовать порядку, указанному в программе, то есть, они не могут переупорядочиваться JIT-компилятором или процессором. Поэтому «изменчивые» (volatile) поля не преподносят неприятных сюрпризов и поддерживают возможность использования в многопоточной среде, но они являются нескоординированными, поэтому сохраняется риск попасть в ситуацию гонки за ресурсами.

С другой стороны, несинхронизированные поля являются «обычными» изменяемыми полями Java, использовать которые в многопоточной среде выполнения можно только под защитой блокировок[2]... или при умелом подходе к ним.

Неизменяемые поля объявляются общедоступными, но изменяемые поля всегда определяются как приватные (private) и доступны только в методах, встроенных в объявление типа. Подробнее о встроенных реализациях будет рассказываться в разделе «Реализация протоколов», ниже, а пока достаточно будет познакомиться с простым примером объявления типа, содержащего единственное изменяемое поле и единственный метод, реализующий интерфейс абстракции `deref`, `IDeref`[3]:

---

[1] Согласно модели памяти в Java, запись в поля `long` или `double`, объявленные без модификатора `volatile`, может выполняться не атомарно. См. JSR-133, раздел 11.

[2] Достаточно храбрые из вас, кто нуждается в большом количестве изменяемых полей, могут использовать макрос `locking`, описанный в разделе «Блокировки» главы 3.

[3] Любой тип, реализующий интерфейс `IDeref`, может разыменовываться с помощью `deref` и, соответственно, с оператора `@`, как описывается в примечании, в разделе «delay» главы 4.

Кот Шредингера

```
(deftype SchrödingerCat [^:unsynchronized-mutable state]
  clojure.lang.IDeref
  (deref [sc]
    (locking sc
      (or state
        (set! state (if (zero? (rand-int 2))
                      :dead
                      :alive))))))

(defn schrödinger-cat
  "Создает нового кота Шредингера. Осторожно, REPL может убить его!"
  []
  (SchrödingerCat. nil))

(def felix (schrödinger-cat))
;= #'user/felix                    ❶
@felix                              ❷
;= :dead
(schrödinger-cat)
;= #<SchrödingerCat@3248bc64: :dead>
(schrödinger-cat)
;= #<SchrödingerCat@3248bc64: :alive>
```

❶ Кот Феликс одновременно и жив, и мертв...

❷ ...пока мы или REPL не убьем его (или не убедимся, что он жив) в результате побочного эффекта разыменования.

Внимание. Изменяемые поля или ссылочные/потоковые типы? Поскольку состояние `SchrödingerCat` не определено, пока мы не «заглянем в контейнер», оно эквивалентно:

```
(delay (if (zero? (rand-int 2))
         :dead
         :alive))
```

В большинстве случаев потребность в изменяемых данных может быть удовлетворена с помощью ссылочных типов (агентов, атомов и ссылок), потоковых типов (`future`, `promise` и `delay`) или за счет разумного использования классов из пакета `java.util.concurrent`. Эти средства (все они описываются в главе 4) являются гораздо более предпочтительными, чем изменяемые поля: они ликвидируют массу сложностей, ограничивают свободу изменений узким кругом мест, и вообще не позволят вам причинить проблемы самим себе. Проще говоря, не стремитесь заполучить изменяемые поля, когда в ваших руках имеются более подходящие инструменты!

Как было показано выше, записать новое значение в изменяемое поле можно с помощью простой формы `set!`, передав ей имя поля, указанное в векторе с полями макроса `deftype`, и новое значение, то есть, `(set! field value)`[1]. Прочитать значение изменяемого поля в теле встроенного метода, можно просто обратившись к нему по его имени[2].

Теперь, когда мы разобрались с различными видами типов в языке Clojure, можно продолжить знакомство с протоколами.

Реализация протоколов

Существует два способа реализации протокола для заданного типа:

1. Включить реализацию методов протокола в определение `deftype` или `defrecord` – это называется *встроенной реализацией* (inline implementation).
2. Задействовать функции `extend*` и с их помощью зарегистрировать методы протокола для типа.

Сравним эти два способа, реализовав протокол `Matrix` для типа записи `Point`, как показано в примерах 6.3 и 6.4, где предполагается, что точка является матрицей размера 2×1:

Пример 6.3. Встроенная реализация протокола

```
(defrecord Point [x y]
  Matrix
  (lookup [pt i j]
    (when (zero? j)
      (case i
        0 x
        1 y)))
  (update [pt i j value]
    (if (zero? j)
      (condp = i
```

[1] Что эквивалентно выражению `this.field = value` в Java, `self.field = value` в Python и `@field = value` в Ruby.

[2] Изменяемые поля внутри встроенных методов действуют так же, как локальные привязки, то есть, к ним можно обращаться непосредственно, например: `(.x this)`. Кроме того, их имена можно «замаскировать» как «настоящие» локальные привязки, определив такие же имена в форме `let` или в векторе аргументов функции.

```
        0 (Point. value y)
        1 (Point. x value))
      pt))
  (rows [pt] [[x] [y]])
  (cols [pt] [[x y]])
  (dims [pt] [2 1]))
```

Пример 6.4. Добавление поддержки протокола к уже определенному типу

```
(defrecord Point [x y])

(extend-protocol Matrix
  Point
  (lookup [pt i j]
    (when (zero? j)
      (case i
        0 (:x pt)
        1 (:y pt))))
  (update [pt i j value]
    (if (zero? j)
      (condp = i
        0 (Point. value (:y pt))
        1 (Point. (:x pt) value))
      pt))
  (rows [pt]
    [[(:x pt)] [(:y pt)]])
  (cols [pt]
    [[(:x pt) (:y pt)]])
  (dims [pt] [2 1]))
```

Одно из незначительных отличий между двумя подходами заключается в особенностях доступа к значениям полей: когда поддержка протокола добавляется к типу извне, обращение к полям типа производится как к ключевым словам (или с использованием формы вида `(.x pt)`), а во встроенных методах к значениям полей можно обращаться непосредственно, по их именам, потому что они находятся в лексической области видимости.

Помимо этого между встроенной реализацией и реализацией с использованием функций `extend-*` существует множество отличий.

Встроенная реализация

Вообще говоря, встроенные методы обладают более высокой производительностью по двум причинам: они могут напрямую об-

ращаться к полям типа, а их вызовы выполняются так же быстро, как вызовы методов интерфейсов в Java.

Поскольку каждый протокол в конечном итоге превращается в интерфейс Java, добавление встроенных реализаций методов в протокол приводит к созданию класса, реализующего этот интерфейс, методы которого являются встроенными реализациями. Кроме того, в каждой точке вызова функции протокола сначала выполняется проверка на соответствие интерфейсу, в результате которой выбирается наиболее быстрый путь, если он доступен: вызов обычного метода, доступный виртуальной машине JVM и высоко оптимизированный.

Однако вся эта машинерия может отказывать при использовании встроенных методов, из-за конфликтов между методами протоколов с одинаковыми именами и сигнатурами. Нельзя определить встроенные методы с одинаковыми именами и сигнатурами, реализующие два разных протокола. Нельзя даже определить встроенные методы одного протокола, чьи сигнатуры совпадают с какими-либо методами `java.lang.Object`. Или, еще хуже, нельзя определить встроенные методы, имена и сигнатуры которых совпадают с любыми методами интерфейсов, автоматически добавляемых макросом `defrecord`, включая `java.util.Map`, `java.io.Serializable`, `clojure.lang.IPersistentMap` и другие. В таких случаях генерируется сообщение об ошибке:

```
(defprotocol ClashWhenInlined
  (size [x]))
;= ClashWhenInlined

(defrecord R []
  ClashWhenInlined
  (size [x]))                          ❶
;= #<CompilerException java.lang.ClassFormatError:
;=    Duplicate method name&signature in class file user/R,
;=    compiling:(NO_SOURCE_PATH:1)>

(defrecord R [])
;= user.R
(extend-type R                         ❷
  ClashWhenInlined
  (size [x]))
;= nil
```

❶ Конфликтует с методом `size` интерфейса `java.util.Map`, реализация которого автоматически добавляется в записи.

❷ Однако при использовании `extend-type` для регистрации методов никаких конфликтов не возникает, потому что это расширение не влияет на конструкцию типа `R` – оно не изменяет реализацию интерфейса, лежащего в основе протокола.

Так как встроенные расширения сохраняются в типе, создаваемом макросом `deftype` или `defrecord`, их реализацию нельзя изменить во время выполнения, не переопределив тип целиком. А это означает, что потребуется заново пересмотреть весь код, напрямую зависящий от этого типа[1]. Однако самая серьезная проблема встроенных методов состоит в том, что оба макроса, `deftype` и `defrecord`, создают и определяют новый тип, из-за чего существующие объекты, созданные ранее измененного типа, *никогда* не будут использовать любые изменившиеся встроенные методы.

Таким образом, встроенные методы, несмотря на всю их привлекательность из-за того, что они так знакомы (еще бы, это же методы классов!), оказываются более статичными, чем другие механизмы распространения протоколов на типы, и их лучше использовать для оптимизации производительности.

Исключением из этого общего правила является ситуация, когда тип Clojure реализует интерфейс Java, так как встроенные методы являются единственно возможным способом.

Встроенные реализации интерфейсов Java

Встроенные методы протокола являются всего лишь реализацией соответствующего интерфейса Java, поэтому тот же самый подход можно использовать для реализации методов любого интерфейса Java, и, как частный случай, методы `java.lang.Object`[2]. Как и в случае с протоколами, нет необходимости реализовать все методы интерфейса, но при обращении к нереализованным методам будет возбуждаться исключение.

---

[1] Объем такого кода можно существенно уменьшить, если следовать совету, данному в разделе «Конструкторы и фабричные функции», выше, и добавлять реализации фабричных функций к своим типам.

[2] Никакой другой класс, абстрактный или нет, не может быть расширен встроенными методами. Для этого необходимо создать класс, производный от конкретного типа, с помощью макросов `gen-class` или `proxy`, описываемых в разделах «Определение именованных классов» и «Экземпляры анонимных классов: proxy», в главе 9, соответственно.

```
(deftype MyType [a b c]
  java.lang.Runnable
  (run [this] ...)
  Object
  (equals [this that] ...)
  (hashCode [this] ...)
  Protocol1
  (method1 [this ...] ...)
  Protocol2
  (method2 [this ...] ...)
  (method3 [this ...] ...))
```

Возможность добавления новых методов в `Object` позволяет реализовать семантику значения для типа `Point`, даже когда он определяется с помощью `deftype`:

```
(deftype Point [x y]
  Matrix
  (lookup [pt i j]
    (when (zero? j)
      (case i
        0 x
        1 y)))
  (update [pt i j value]
    (if (zero? j)
      (case i
        0 (Point. value y)
        1 (Point. x value))
      pt))
  (rows [pt]
    [[x] [y]])
  (cols [pt]
    [[x y]])
  (dims [pt]
    [2 1])
  Object
  (equals [this other]
    (and (instance? (class this) other)        ❶
      (= x (.x other)) (= y (.y other))))
  (hashCode [this]
    (-> x hash (hash-combine y))))
```

❶ Встроенные методы не могут ссылаться на определяемый тип.

Внимание. Реализация методов `equals` и `hashCode` в Clojure так же требует внимательности и осторожности, как и в Java – автоматическое добавление их реализаций макросом `defrecord` является одной из самых привлекательных его сторон. Однако реализация этих методов для типов, создаваемых с помощью `deftype`, еще более сложная задача, потому что во встроенных методах нельзя ссылаться на определяемый тип: если в предыдущем примере попробовать заменить `(class this)` на `Point`, он скомпилируется, но выражение `(instance? Point other)` всегда будет возвращать `false`. Это – известное ограничение текущего компилятора. В число других обходных решений входят: вызов из встроенного метода функции, которая может сослаться на определяемый класс, или проверка на соответствие интерфейсу (протоколу) вместо конкретного класса.

Определение анонимных типов с помощью reify

Помимо `defrecord` и `deftype` существует еще одна конструкция, принимающая встроенные методы: `reify`.

В отличие от `deftype` и `defrecord`, `reify` не является формой верхнего уровня – вместо определения именованных типов она создает экземпляры анонимных типов. По сути – это способ создания объектов, поддерживающих любой протокол (или реализующих методы любых интерфейсов или класса `Object`). Это делает их аналогами анонимных вложенных классов в Java.

Общая структура определения `reify` идентична структуре определений в `defrecord` и `deftype`, но без объявления полей:

```
(reify
  Protocol-or-Interface-or-Object
  (method1 [this x]
    (implementation))
  Another-Protocol-or-Interface
  (method2 [this x y]
    (implementation))
  (method3 [this x]
    (implementation)))
```

Так же как в случае с типами и записями, здесь не обязательно определять реализации всех методов данного протокола или интерфейса.

Экземпляры, созданные с помощью `reify`, образуют замыкания, обеспечивающие прямой доступ из методов ко всем локальным привязкам, находящимся в лексической области видимости. Эта особенность очень удобна при создании адаптеров (как показано в примере 6.5) или одноразовых экземпляров (как показано в примере 6.6).

Пример 6.5. Превращает функцию в реализацию интерфейса ActionListener

```
(defn listener
  "Создает экземпляр AWT/Swing `ActionListener`, делегирующий выполнение
   операций указанной функции."
  [f]
  (reify
    java.awt.event.ActionListener
    (actionPerformed [this e]
      (f e))))
```

Пример 6.6. Использование реализации FileFilter для получения каталогов

```
(.listFiles (java.io.File. ".")
  (reify
    java.io.FileFilter
    (accept [this f]
      (.isDirectory f))))
```

Для решения подобных задач можно также использовать макрос `proxy`[1], но:

- `reify` проще: реализации методов, объявленные в этой форме, «впекаются» в объект, как это делают формы `deftype` и `defrecord`, поэтому динамические расширения не поддерживаются непосредственно;
- `reify` имеет множество ограничений: эта форма может использоваться только для реализации методов протоколов, интерфейсов Java и класса `Object`; конкретные классы, абстрактные или нет, не могут наследоваться;
- поскольку все методы в форме `reify` встраиваются в класс, который создается за кулисами, накладные расходы на их вызовы равны нулю.

Повторное использование реализаций

В модели протоколов и типов языка Clojure отсутствует понятие иерархии. Как будет показано далее, это нельзя считать ограничением, поскольку наследование на основе типов само по себе является сложным ограничением и Clojure предоставляет более мощную и гибкую альтернативу.

[1] См. раздел «Экземпляры анонимных классов: proxy» в главе 9.

Типы могут только реализовать протоколы или интерфейсы – в Clojure невозможно организовать наследование типов, как во многих других языках, где допускается определять новые типы, являющиеся подклассами других конкретных типов, и тем самым наследующие реализации методов. Для повторного использования конкретных реализаций методов в Clojure предоставляется функция `extend`, являющаяся основой макросов `extend-type` и `extend-protocol`.

`extend` принимает в первом аргументе расширяемый тип, а в последующих – череду протоколов и *карт реализаций* (implementation maps), которые отображают имена (как ключевые слова) в функции, реализующие методы с этими именами для указанного типа.

Воспользуемся функцией `extend`, чтобы добавить поддержку протокола `Matrix` в тип записи `Point`:

```
(defrecord Point [x y])

(extend Point
  Matrix
  {:lookup (fn [pt i j]
             (when (zero? j)
               (case i
                 0 (:x pt)
                 1 (:y pt))))
   :update (fn [pt i j value]
             (if (zero? j)
               (condp = i
                 0 (Point. value (:y pt))
                 1 (Point. (:x pt) value))
               pt))
   :rows (fn [pt]
           [[(:x pt)] [(:y pt)]])
   :cols (fn [pt]
           [[(:x pt) (:y pt)]])
   :dims (fn [pt] [2 1]))
```

Так как `extend` – это функция, а не макрос, подобно `extend-type` и `extend-protocol`, карта реализаций интерпретируется как *значение*, которое можно передавать, изменять и комбинировать с другими картами реализаций. Благодаря такой гибкости, можно моделировать любые варианты повторного использования, от простого «наследования» до более сложных понятий, таких как трейты (traits) и примеси (mixins).

В качестве простого примера можно привести реализацию по умолчанию методов rows и cols, зависящую только от конкретных реализаций методов dims и lookup:

```
(def abstract-matrix-impl
  {:cols (fn [pt]
           (let [[h w] (dims pt)]
             (map
               (fn [x] (map #(lookup pt x y) (range 0 w)))
               (range 0 h))))
   :rows (fn [pt]
           (apply map vector (cols pt)))})
```

Теперь можно добавить поддержку протокола Matrix в тип Point, опираясь на реализации по умолчанию, просто добавляя их в карту реализаций с помощью assoc:

```
(extend Point
  Matrix
  (assoc abstract-matrix-impl
    :lookup (fn [pt i j]
              (when (zero? j)
                (case i
                  0 (:x pt)
                  1 (:y pt))))
    :update (fn [pt i j value]
              (if (zero? j)
                (condp = i
                  0 (Point. value (:y pt))
                  1 (Point. (:x pt) value))
                pt))
    :dims (fn [pt] [2 1])))
```

Несмотря на простоту и отсутствие чего-либо нового, с точки зрения проектирования классов, этот пример наглядно иллюстрирует как можно моделировать отношения наследования, не зависящие от конкретных типов, благодаря возможности интерпретировать методы протоколов как обычные именованные функции.

Гораздо интереснее выглядит возможность использования реализаций как значений для создания *примесей* (mixins), то есть возможность комбинировать различные реализации одного и того же протокола некоторым значимым способом.

Например, попробуем определить новый протокол, `Measurable`[1]:

```
(defprotocol Measurable
  "Протокол для извлечения размеров виджетов (widgets)."
  (width [measurable] "Returns the width in px.")
  (height [measurable] "Returns the height in px."))
```

Затем определим новый тип записи `Button`, в который добавим поддержку протокола `Measurable`, а также карту реализаций `bordered`:

```
(defrecord Button [text])

(extend-type Button
  Measurable
  (width [btn]
    (* 8 (-> btn :text count)))
  (height [btn] 8))

(def bordered
  {:width #(* 2 (:border-width %))
   :height #(* 2 (:border-height %))})
```

Теперь можно определить тип `BorderedButton`, объединяющий реализации `Button` и `bordered`. Но здесь есть одна проблема: у нас нет карты реализаций для `Button`. Протоколы – это не просто статические имена, которые можно использовать с функцией `extend`, и другими; это имена переменных, содержащих массу полезного:

Пример 6.7. Содержимое карты реализаций переменой-протокола

```
Measurable
;= {:impls
;=   {user.Button
;=     {:height #<user$eval2056$fn__2057 user$eval2056$fn__2057@112f8578>,
;=      :width #<user$eval2056$fn__2059 user$eval2056$fn__2059@74b90ff7>}},
;=  :on-interface user.Measurable,
;=  :on user.Measurable,
;=  :doc "Протокол для извлечения размеров виджетов (widgets).",
```

[1] Этот пример появился под влиянием статьи о примесях в Википедии: https://en.wikipedia.org/wiki/Mixin (аналогичную, хотя и не такую подробную, статью на русском языке можно найти по адресу: https://ru.wikipedia.org/wiki/Mixin – *Прим. перев.*).

```
;=  :sigs
;=    {:height
;=       {:doc "Возвращает высоту в пикселях.",
;=        :arglists ([measurable]),
;=        :name height},
;=     :width
;=        {:doc "Возвращает ширину в пикселях.",
;=         :arglists ([measurable]),
;=         :name width}},
;=  :var #'user/Measurable,
;=  :method-map {:width :width, :height :height},
;=  :method-builders
;=    {#'user/height #<user$eval2012$fn__2013
;=        user$eval2012$fn__2013@27aa7aac>,
;=     #'user/width #<user$eval2012$fn__2024
;=        user$eval2012$fn__2024@4848268a>}}
```

Здесь можно увидеть массу интересного – большинство (если не все) деталей реализации[1]. Обратите внимание, что для нашего примера примеси выражение `(get-in Measurable [:impls Button])` возвращает карту реализаций протокола `Measurable` в типе `Button`:

```
(get-in Measurable [:impls Button])
;= {:height #<user$eval1251$fn__1252 user$eval1251$fn__1252@744589eb>,
;=  :width #<user$eval1251$fn__1254 user$eval1251$fn__1254@40735f45>}
```

Нам не хватает функции, объединяющей различные реализации некоторого метода в новую реализацию. Такая функция должна принимать функции для объединения и еще одну функцию, вычисляющую новый результат на основе результатов использования двух других реализаций:

```
(defn combine
  "Принимает две функции, f и g, и возвращает fn, которая принимает список
   аргументов переменной длины, применяет их к f и g, и возвращает результат
   (op rf rg), где rf и rg – результаты вызовов f и g."
  [op f g]
  (fn [& args]
    (op (apply f args) (apply g args))))
```

[1] Краткий обзор функций механизма интроспекции протоколов можно найти в разделе «Интроспекция протоколов», ниже.

Наконец можно определить тип BorderedButton и добавить в него поддержку протокола Measurable, используя + для объединения результатов, полученных с помощью методов из карты реализаций bordered, и методов протокола Measurable, уже зарегистрированных в типе Button:

```
(defrecord BorderedButton [text border-width border-height])

(extend BorderedButton
  Measurable
  (merge-with (partial combine +)
    (get-in Measurable [:impls Button])
    bordered))
```

Пришло время убедиться, что экземпляр типа BorderedButton правильно вычисляет размеры, в сравнении с экземпляром типа Button, с тем же текстом надписи:

```
(let [btn (Button. "Hello World")]
  [(width btn) (height btn)])
;= [88 8]

(let [bbtn (BorderedButton. "Hello World" 6 4)]
  [(width bbtn) (height bbtn)])
;= [100 16]
```

Примечание. Еще один аргумент против использования встроенных методов состоит в том, что такие мощные шаблоны повторного использования реализаций методов недоступны для типов, включающих встроенные расширения. В этих случаях вам придется обратиться к приему делегирования и/или использовать макросы.

Интроспекция протоколов

Теперь, когда мы увидели в примере 6.7 внутреннее устройство протоколов, можно перечислить вспомогательные функции для интроспекции протоколов: extenders, extends? и satisfies?, которые все вместе образуют прикладной интерфейс, помогающий получить ответы на часто задаваемые вопросы о протоколах и их взаимоотношениях с типами.

`extenders`

Возвращает классы, расширяемые данным протоколом. Например, после выполнения примеров из предыдущего раздела, связанных с протоколом `Measurable`, можно увидеть, какие типы поддерживают этот протокол:

```
(extenders Measurable)
;= (user.BorderedButton user.Button)
```

Это все равно, что спросить: «Какие классы реализуют Java-интерфейс `x`?». Обратите внимание: так как поддержку протоколов можно добавить в типы *в любой момент во время выполнения*, эти результаты действительны только на момент вызова функции `extenders`.

`extends?`

Возвращает `true`, только если тип поддерживает протокол:

```
(extends? Measurable Button)
;= true
```

`satisfies?`

Аналог функции `instance?`: проверяет, удовлетворяет ли конкретный экземпляр указанному протоколу, либо за счет добавления поддержки посредством макроса `extend` и его производных:

```
(satisfies? Measurable (Button. "hello"))
;= true
(satisfies? Measurable :other-value)
;= false
```

...либо благодаря наличию в типе встроенной реализации методов протокола:

```
(deftype Foo [x y]
  Measurable
  (width [_] x)
  (height [_] y))
;= user.Foo
(satisfies? Measurable (Foo. 5 5))
;= true
```

В последнем случае, поскольку типы, имеющие встроенные реализации, в действительности всего лишь реализуют методы интерфейса, генерируемого протоколом, функции `satisfies?` и `instance?` могут действовать взаимозаменяемо:

```
(instance? user.Measurable (Foo. 5 5))     ❶
;= true
```

❶ `user.Measurable` – это интерфейс, сгенерированный протоколом `Measurable`, который мы определили в `#'user/Measurable`.

Пограничные случаи использования протоколов

Поскольку для методов протоколов создаются функции, привязанные к определенному пространству имен, функции двух протоколов никогда не конфликтуют между собой. Однако есть некоторые пограничные случаи, несвойственные типичным объектно-ориентированным языкам, когда использование протоколов может давать неожиданные результаты.

Соперничающие реализации. Возможность включения поддержки протокола в любой момент во время выполнения является огромным благом с точки зрения разработки в интерактивном режиме и позволяет развивать реализацию протокола, быстро устраняя проблемы, связанные с моделированием и оптимизацией[1]. Однако, если для некоторого типа существует две реализации одного и того же протокола, тогда последняя загруженная реализация заменит первую, что может приводить к неожиданным эффектам, если ожидаемая реализация загружается первой.

Увы, это – не техническая проблема, а организационная! Если вы не являетесь разработчиком ни протокола, ни типа, тогда, чтобы разрешить подобные конфликты, *готовьтесь создать собственную реализацию*. Потенциальный конфликт между двумя разработчиками обычно разрешается в хронологическом порядке: если протокол создается раньше типа, ответственность за поддержку протокола ложится на плечи разработчика типа, и наоборот.

Иерархии классов нарушают «связи». Похожая ситуация, когда существует две реализации одного протокола для двух родственных

[1] Пример устранения таких проблем можно найти в статье по адресу: http://dosync.posterous.com/51626638.

типов, и для вызова определенного метода протокола может использоваться любой из двух типов. Например, допустим, что мы распространили протокол на два родственных интерфейса, `java.util.List` и `java.util.Collection`, и затем вызываем методы протокола для типа, соответствующего обеим версиям протокола:

```
(defprotocol P
  (a [x]))
;= P
(extend-protocol P
  java.util.Collection
  (a [x] :collection!)
  java.util.List
  (a [x] :list!))
;= nil
(a [])
;= :list!
```

В таких случаях выбор реализации протокола производится исходя из отношения родства в иерархии классов – всегда выбирается реализация протокола для более специализированного типа, которым в данном случае является `java.util.List`, наследующий базовый интерфейс `Collection`.

Отсутствие «связи» ведет к выбору произвольной реализации. А что произойдет, если поддержка протокола будет распространена на два типа, не состоящих в родстве? В этом случае механизм выбора реализации протокола выберет и запомнит *произвольную* реализацию. В качестве примера наиболее вероятной ситуации такого рода можно привести распространение протокола на несколько высокоуровневых интерфейсов Java.

Рассмотрим протокол с двумя реализациями метода, поддержка которого включается в два разных интерфейса, не связанных отношением родства, `java.util.Map` и `java.io.Serializable`:

```
(defprotocol P
  (a [x]))

(extend-protocol P
  java.util.Map
  (a [x] :map!)
  java.io.Serializable
  (a [x] :serializable!))
```

Что произойдет, если вызвать метод `a` для ассоциативного массива, который в языке Clojure поддерживает оба этих интерфейса?

```
(a {})
;= :serializable!
```

Результат получился вполне осмысленный, но почему была выбрана эта реализация, а не другая, для типа `Map`? Проблема в том, что выбор реализации протокола для каждого конкретного типа при вызове одного и того же метода может измениться при следующем запуске приложения или REPL. Для примера выше это означает, что заранее неизвестно, что вернет вызов `(a {})`, `:serializable!` или `:map!`.

Существует несколько способов решения этой проблемы:

1. Распространяйте протоколы только на конкретные типы, которые действительно должны поддерживать эти протоколы. Конкретные типы всегда однозначны[1].
2. Желание распространить протокол на несколько несвязанных высокоуровневых интерфейсов может служить признаком проблемы проектирования, когда контракт протокола оказывается слишком обширным. Пересмотрите этот контракт.
3. Используйте мультиметоды. В отличие от протоколов, мультиметоды генерируют ошибку, когда невозможно однозначно определить вызываемую реализацию. Кроме того, мультиметоды включают механизм «привилегий», позволяющий определить порядок разрешения подобных неоднозначностей[2].

Поддержка абстракций коллекций

А разделе «Кот Шредингера» (выше) мы мельком видели, как можно обеспечить поддержку одной из абстракций Clojure пользовательским типом, объявив тип разыменовываемым (dereferenceable)[3]. Теперь подымем планку еще выше и определим структуру данных, обеспечивающую полную поддержку абстракций Clojure: множество на основе массива, более эффективное при небольшом

---

[1] ...И, соответственно, однозначен выбор реализации протокола для типов, объявленных с помощью `deftype` или `record`.

[2] Мультиметоды подробно описываются в главе 7, а механизм привилегий — в разделе «Множественное наследование», в той же главе.

[3] Реализовав в нем метод `deref`. — *Прим. перев.*

количестве элементов, как в смысле производительности, так и в смысле расходования памяти, чем стандартные множества на основе деревьев[2].

Поддержка абстракции языка Clojure в настоящее время означает необходимость реализации интерфейсов Java, которые определяются языком Clojure для каждой абстракции. То есть, поддержка абстракций может быть реализована только с помощью встроенных методов[2].

Самое сложное в организации поддержки абстракций заключается в выяснении интерфейсов и методов, которые требуется реализовать, так как все это в значительной степени не документировано официально[3]. Значительную помощь в этом отношении может оказать следующая вспомогательная функция:

```
(defn scaffold
  "Для заданного интерфейса возвращает 'полое' тело, которое можно
   использовать в определении `deftype`."
  [interface]
  (doseq [[iface methods] (->> interface
                               .getMethods
                               (map #(vector (.getName (.getDeclaringClass %))
                                             (symbol (.getName %))
                                             (count (.getParameterTypes %))))
                               (group-by first))]
    (println (str " " iface))
    (doseq [[_ name argcount] methods]
       (println
         (str " "
           (list name (into '[this]
                            (take argcount (repeatedly gensym))))))))
```

---

[1] Стратегия реализации на основе деревьев, используемая в Clojure для большинства коллекций и описанная в разделе «Визуализация сохранности: ассоциативные массивы (векторы и множества)», в главе 3, хорошо зарекомендовала себя на практике, но в особых случаях может потребоваться создать специализированные реализации структур данных.

[2] В будущем интерфейсы основных абстракций планируется заменить протоколами. Такая замена уже выполнена в ClojureScript, описываемом в разделе «ClojureScript», в главе 20.

[3] Clojure Atlas – один из инструментов, который поможет определить, какие интерфейсы стоят позади абстракций языка Clojure: http://www.clojureatlas.com.

Взглянув на вывод, полученный в результате вызова `(ancestors (class #{}))`, можно увидеть, что `clojure.lang.IPersistentSet` является основным интерфейсом из числа реализуемых множествами в языке Clojure, Передав его функции `scaffold`, вы получите отличную заготовку для собственной реализации множества:

```
(scaffold clojure.lang.IPersistentSet)
;  clojure.lang.IPersistentSet
;    (get [this G__5617])
;    (contains [this G__5618])
;    (disjoin [this G__5619])
;  clojure.lang.IPersistentCollection
;    (count [this])
;    (cons [this G__5620])
;    (empty [this])
;    (equiv [this G__5621])
;  clojure.lang.Seqable                    ❶
;    (seq [this])
;  clojure.lang.Counted
;    (count [this])                        ❷
```

❶ Если необходимо, чтобы структура данных поддерживала возможность преобразования в последовательность, достаточно реализовать метод `seq` интерфейса `clojure.lang.Seqable`.

❷ Обратите внимание на наличие двух методов `count` с одинаковой сигнатурой; от одного из них придется отказаться.

Чтобы создать собственный тип множества и гарантировать поддержку семантики значения, нам необходимо реализовать эти методы, вдобавок к методам `hashCode` и `equals` типа `Object`. Назначение всех этих методов должно быть очевидно, за исключением, разве что, `cons` и `equiv`. Метод `cons`, несмотря на свое название, является методом, на который опирается функция `conj`; `equiv` является опорой для функции `equals`, но реализует адекватную семантику равенства, применимую к числам (см. раздел «Эквивалентность может защитить ваш рассудок» в главе 11). К последнему мы не будем предъявлять специальные требования, так как наше множество не является числовым типом.

Пример 6.8. Реализация множества на основе массива с помощью deftype

```
(declare empty-array-set)
(def ^:private ^:const max-size 4)                    ❶
```

```
(deftype ArraySet [^objects items
                   ^int size
                   ^:unsynchronized-mutable ^int hashcode]
  clojure.lang.IPersistentSet
  (get [this x]
    (loop [i 0]
      (when (< i size)
        (if (= x (aget items i))
          (aget items i)
          (recur (inc i))))))
  (contains [this x]
    (boolean
      (loop [i 0]
        (when (< i size)
          (or (= x (aget items i)) (recur (inc i)))))))
  (disjoin [this x]
    (loop [i 0]
      (if (== i size)
        this
        (if (not= x (aget items i))
          (recur (inc i))
          (ArraySet. (doto (aclone items)
                       (aset i (aget items (dec size)))
                       (aset (dec size) nil))
                     (dec size)
                     -1)))))
  clojure.lang.IPersistentCollection
  (count [this] size)
  (cons [this x]
    (cond
      (.contains this x) this
      (== size max-size) (into #{x} this)                          ❷
      :else (ArraySet. (doto (aclone items)
                         (aset size x))
                       (inc size)
                       -1)))
  (empty [this] empty-array-set)
  (equiv [this that] (.equals this that))                          ❸
  clojure.lang.Seqable
  (seq [this] (take size items))
  Object
  (hashCode [this]
    (when (== -1 hashcode)
      (set! hashcode (int (areduce items idx ret 0                 ❹
```

```
                              (unchecked-add-int ret (hash (aget items idx)))))))
    hashcode)
  (equals [this that]
    (or
      (identical? this that)
      (and (or (instance? java.util.Set that)
               (instance? clojure.lang.IPersistentSet that))
           (= (count this) (count that))
           (every? #(contains? this %) that)))))

(def ^:private empty-array-set (ArraySet. (object-array max-size) 0 -1))

(defn array-set                                                          ❺
  "Создает множество на основе массива, содержащее указанные значения."
  [& vals]
  (into empty-array-set vals))
```

❶ Ключевая характеристика реализации коллекции, отражающая особый...

❷ ...характер набора данных, заключается в том, что как только объем данных выходит за «оптимальные границы», реализация должна «преобразовывать» данные в коллекцию другого, более подходящего типа, не выходя за границы рассматриваемой абстракции. В данном случае множество оптимизировано для хранения малого количества значений, поэтому мы преобразуем данные в обычное множество языка Clojure, как только количество значений в множестве становится больше четырех. Для других узкоспециальных случаев более оптимальным может оказаться другое пороговое значение.

❸ `equiv` делегирует свою работу методу `equals`, чтобы привести поведение типа `ArraySet` в соответствие с поведением `clojure.lang.APersistentSet`.

❹ Поскольку реализация данного множества основана на массиве, нам пришлось использовать операции, – `areduce`, `aget`, `aset` и другие – о которых еще не рассказывалось, хотя они не имеют прямого отношения к обсуждаемой теме. Описание операций над массивами можно найти в разделе «Используйте простые массивы осмысленно», в главе 11.

❺ Конструктор типа `ArraySet` никак не годится для конечных пользователей – его поля целиком связаны с особенностями его реализации. По этой и многим другим уважительным причинам, о которых рассказывалось в разделе «Конструкторы и фабричные функции», выше, мы добавили дружественную фабричную функцию `array-set`.

Работает ли эта реализация?

```
(array-set)
;= #{}
(conj (array-set) 1)
;= #{1}
(apply array-set "hello")
;= #{\h \e \l \o}
(get (apply array-set "hello") \w)
;= nil
(get (apply array-set "hello") \h)
;= \h
(contains? (apply array-set "hello") \h)
;= true
(= (array-set) #{})
;= true
```

Пока неплохо, но...

```
((apply array-set "hello") \h)
; #<ClassCastException java.lang.ClassCastException:
;    user.ArraySet cannot be cast to clojure.lang.IFn>
```

В этом исключении нет ничего неожиданного: мы пока не предусмотрели возможность использования экземпляров `ArraySet` в качестве функций. Для этого нужно добавить поддержку интерфейса `clojure.land.IFn`, который реализуют все функции в языке `Clojure`. В целом, это необязательно – нигде не говорится, что пригодная к использованию коллекция *должна* поддерживать возможность ее вызова как функции – но данная возможность весьма удобна, поэтому мы возьмем на себя труд добавить ее[1].

Более серьезным выглядит следующий недостаток:

```
(= #{} (array-set))
;= false
```

Мы нарушили свойство симметричности оператора `=`, ключевую часть его контракта. Это обусловлено тем, что по определению

---

[1] Точно так же тип `ArraySet` пока не поддерживает метаданные – функциональность, определяемую интерфейсом `clojure.lang.IObj`. Он определяет всего два очень простых метода, поэтому их реализацию оставим читателям, в качестве самостоятельного упражнения.

множества Clojure одновременно являются множествами Java, как определено `java.util.Set`[1].

Воспользуемся функцией `scaffold` еще раз, чтобы посмотреть, какие методы нужно добавить, чтобы реализовать поддержку `java.util.Set`:

```
(scaffold java.util.Set)
;  java.util.Set
;    (add [this G__6140])
;    (equals [this G__6141])
;    (hashCode [this])
;    (clear [this])
;    (isEmpty [this])
;    (contains [this G__6142])
;    (addAll [this G__6143])
;    (size [this])
;    (toArray [this G__6144])
;    (toArray [this])
;    (iterator [this])
;    (remove [this G__6145])
;    (removeAll [this G__6146])
;    (containsAll [this G__6147])
;    (retainAll [this G__6148])
```

Не пугайтесь: реализовать необходимо только часть интерфейса, потому что тип `ArraySet` является неизменяемым. Методы `equals`, `hashCode` и `contains` уже готовы, поэтому остаются только:

```
java.util.Set
  (isEmpty [this])
  (size [this])
  (toArray [this G__6144])
  (toArray [this])
  (iterator [this])
  (containsAll [this G__6147])
```

Все эти методы достаточно просты. Единственная ловушка – утечка массива `items` при возврате из `toArray`; это может сделать возможным внесение изменений во внутренний массив, в нарушение

[1] То же относится и к реализациям ассоциативных массивов Clojure, которые ко всему прочему должны реализовать интерфейс `java.util.Map`.

гарантий неизменяемости. Создание метода `iterator` может оказаться утомительным занятием, если попытаться реализовать его в лоб, но очень простым, если помнить, что помимо всего прочего последовательности являются также коллекциями Java и можно просто повторно использовать их реализацию интерфейса `Iterator`:

Пример 6.9. Улучшенная реализация множества на основе массива с использованием deftype

```
(deftype ArraySet [^objects items
                   ^int size
                   ^:unsynchronized-mutable ^int hashcode]
  clojure.lang.IPersistentSet
  (get [this x]
    (loop [i 0]
      (when (< i size)
        (if (= x (aget items i))
          (aget items i)
          (recur (inc i))))))
  (contains [this x]
    (boolean
      (loop [i 0]
        (when (< i size)
          (or (= x (aget items i)) (recur (inc i)))))))
  (disjoin [this x]
    (loop [i 0]
      (if (== i size)
        this
        (if (not= x (aget items i))
          (recur (inc i))
          (ArraySet. (doto (aclone items)
                       (aset i (aget items (dec size)))
                       (aset (dec size) nil))
                     (dec size)
                     -1)))))
  clojure.lang.IPersistentCollection
  (count [this] size)
  (cons [this x]
    (cond
      (.contains this x) this
      (== size max-size) (into #{x} this)
      :else (ArraySet. (doto (aclone items)
                         (aset size x))
                       (inc size)
```

```
                      -1)))
  (empty [this] empty-array-set)
  (equiv [this that] (.equals this that))
  clojure.lang.Seqable
  (seq [this] (take size items))
  Object
  (hashCode [this]
    (when (== -1 hashcode)
      (set! hashcode (int (areduce items idx ret 0
                            (unchecked-add-int ret (hash (aget items idx)))))))
    hashcode)
  (equals [this that]
    (or
      (identical? this that)
      (and (instance? java.util.Set that)                                    ❶
           (= (count this) (count that))
           (every? #(contains? this %) that))))
  clojure.lang.IFn
  (invoke [this key] (.get this key))                                        ❷
  (applyTo [this args]                                                       ❸
    (when (not= 1 (count args))
      (throw (clojure.lang.ArityException. (count args) "ArraySet")))
    (this (first args)))
  java.util.Set
  (isEmpty [this] (zero? size))
  (size [this] size)
  (toArray [this array]
    (.toArray ^java.util.Collection (sequence items) array))                 ❹
  (toArray [this] (into-array (seq this)))
  (iterator [this] (.iterator ^java.util.Collection (sequence this)))        ❺
  (containsAll [this coll]
    (every? #(contains? this %) coll)))

(def ^:private empty-array-set (ArraySet. (object-array max-size) 0 -1))
```

❶ `equals` можно упростить и выполнять проверку, только если `that` реализует интерфейс `java.util.Set`, который теперь поддерживается типом `ArraySet`.

❷ Для поддержки прямых вызовов, `clojure.lang.IFn` определяет метод `invoke`, который может принимать до 21 разных аргументов. Множества могут принимать только один аргумент (искомый ключ, как если бы вызывался метод `get`), поэтому мы реализуем только этот вариант метода; если экземпляр `ArraySet` попытаться вызвать с большим числом аргументов, будет сгенерирована ошибка.

❸ Для поддержки вызова с помощью apply, IFn определяет метод applyTo, который принимает последовательность аргументов, передаваемых функции apply.
❹ Функция sequence предпочтительнее функции seq, потому что никогда не возвращает nil: если для пустого множества seq вернет nil, это вызовет исключение NullPointerException.
❺ Так как последовательности являются коллекциями Java, мы можем просто вернуть Iterator этой коллекции.

Теперь у нас имеется законченная реализация множества, прекрасно уживающегося с другими множествами и поддерживающего возможность вызова, как и обычные множества в Clojure:

```
(= #{3 1 2 0} (array-set 0 1 2 3))
;= true
((apply array-set "hello") \h)
;= \h
```

Но, дает ли тип ArraySet какие-нибудь преимущества? Сравним его производительность с производительностью множества hash-set, используя смесь операций поиска, disj и conj:

```
(defn microbenchmark
  [f & {:keys [size trials] :or {size 4 trials 1e6}}]
  (let [items (repeatedly size gensym)]
    (time (loop [s (apply f items)
                 n trials]
            (when (pos? n)
              (doseq [x items] (contains? s x))
              (let [x (rand-nth items)]
                (recur (-> s (disj x) (conj x)) (dec n))))))))

(doseq [n (range 1 5)
        f [#'array-set #'hash-set]]
    (print n (-> f meta :name) ": ")
    (microbenchmark @f :size n))
; size 1 array-set : "Elapsed time: 839.336 msecs"
; size 1 hash-set : "Elapsed time: 1105.059 msecs"
; size 2 array-set : "Elapsed time: 1201.81 msecs"
; size 2 hash-set : "Elapsed time: 1369.192 msecs"
; size 3 array-set : "Elapsed time: 1658.36 msecs"
; size 3 hash-set : "Elapsed time: 1740.955 msecs"
; size 4 array-set : "Elapsed time: 2197.424 msecs"
; size 4 hash-set : "Elapsed time: 2154.637 msecs"
```

Результаты показывают, что для очень маленьких множеств `array-set` показывает одинаковую или лучшую производительность, чем `hash-set`, а поскольку данная реализация основана на простом массиве Java, для хранения одного и того же объема данных расходуется меньше памяти, чем при использовании `hash-set`.

В заключение

Типы, записи и протоколы, вместе взятые, образуют мощный фундамент, позволяющий сконцентрировать все внимание на данных и избежать лишних сложностей. Такой подход к реализации типов и абстракций позволяет более точно моделировать предметные области и взаимодействия, создавая целые языки для таких моделей – как например возможность работы с записями и ассоциативными массивами единообразным способом с помощью абстракций и функций обычных и ассоциативных коллекций – и помогает избежать ошибчного выбора из-за часто ненужных сложностей, таких как определение иерархий классов.

Глава 7. Мультиметоды

Выше мы говорили о протоколах: они вводят часто используемую, но ограниченную форму полиморфизма – с выбором метода на основе типа. В этой главе мы исследуем *мультиметоды* (multimethods), позволяющие выбирать реализацию не только на основе типа аргумента, но на некоторых других показателях, никак не связанных с типами. То есть, выбор той или иной реализации мультиметода может быть организован, как функция от любого свойства аргумента, без каких-либо привилегий одних перед другими. Кроме того, мультиметоды поддерживают произвольные иерархии и предоставляют различные способы устранения неоднозначностей при множественном наследовании.

> **Примечание.** В Java одно имя может быть присвоено нескольким методам с разными сигнатурами одинаковой длины, отличающимися только типами аргументов. Этот прием называется перегрузкой (overloading). Однако это не означает, что метод имеет несколько реализаций: выбор сигнатуры выполняется на этапе компиляции, исходя из типов аргументов метода. Динамический выбор выполняется только на основе типа привилегированного аргумента `this`.

Основы мультиметодов

Мультиметоды создаются с помощью формы `defmulti`, а реализации мультиметода определяются с помощью форм `defmethod`. Порядок следования определений соответствует порядку следования слов в термине *мультиметод*: сначала определяется поддержка выбора из множества (мульти), а затем – методы, из числа которых осуществляется выбор.

Рассмотрим небольшой пример: функцию, заполняющую узлы XML/HTML, поведение которой зависит от имени тега, использующую модель представления XML, реализованную в пространстве имен `clojure.xml`. Элементом в этой модели является ассоциативный массив с тремя ключами: `:tag` – имя тега как ключевое словл, `:attrs` – ассоциативный массив с именами атрибутов (в виде ключевых

слов) и значениями (в виде строк) и `:content` – коллекция дочерних узлов и содержимого.

```
(defmulti fill
  "Заполняет узел xml/html (согласно модели clojure.xml)
   указанным значением."
  (fn [node value] (:tag node)))                    ❶

(defmethod fill :div                                ❷
  [node value]
  (assoc node :content [(str value)]))

(defmethod fill :input
  [node value]
  (assoc-in node [:attrs :value] (str value)))
```

❶ Это – *функция выбора* (dispatch function); она принимает аргументы мультиметода и возвращает *значение выбора* (dispatch value), которое используется для выбора того или иного метода для данного набора аргументов.

❷ Здесь `:div` – это значение выбора. Если функция выбора вернет значение, совпадающее с этим значением выбора, будет выбран и вызван этот метод (который является всего лишь обычной функцией).

Принцип действия мультиметода прост:

1. Он принимает аргументы.
2. Вычисляет значение выбора, вызывая функцию выбора с данными аргументами.
3. Выбирает тот или иной метод, соответствующий значению выбора.
4. Вызывает выбранный метод с оригинальными аргументами.

Вот и все! Эту схему легко можно реализовать самому с использованием атомов и макросов, уместив ее в несколько строк кода[1], или чуть больше, на языке Ruby или Python[2].

---

[1] Оставляем это читателям в качестве самостоятельного упражнения.

[2] Похожие схемы, напоминающие мультиметоды в Clojure, существуют и в других языках; одной из наиболее зрелых реализаций является PEAK-Rules (http://pypi.python.org/pypi/PEAK-Rules) для Python, написанная Филипом Дж. Эби (Philip J. Eby). Статью с описанием особенностей реализации мультиметодов в стиле Clojure на языке Python можно найти по адресу: http://codeblog.dhananjaynene.com/2010/08/clojure-style-multi-methods-in-python/.

Определение мультиметода имеет два основных отличия от определений обычных функций:

- мультиметод, несмотря на то, что определяет функцию, не содержит явного определения аргументов – он поддерживает все аргументы, которые поддерживаются функциями, доступными для выбора;
- форма `defmulti` в действительности определяет новую переменную (`fill` – в примере выше), а каждая форма `defmethod` просто регистрирует новую реализацию в «корневом» мультиметоде – как это ни странно, но `defmethod` не определяет и не переопределяет никаких переменных.

Пока мы еще не опробовали свой код. Давайте проверим, действительно ли он действует, как ожидалось:

```
(fill {:tag :div} "hello")
;= {:content ["hello"], :tag :div}
(fill {:tag :input} "hello")
;= {:attrs {:value "hello"}, :tag :input}
(fill {:span :input} "hello")
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   No method in multimethod 'fill' for dispatch value: null>
```

Одним из недостатков нашего мультиметода является отсутствие реализации по умолчанию, поэтому `fill` работает только с известными элементами. То есть, для которых определены реализации с помощью `defmethod`.

К счастью, существует специальное значение выбора `:default`.

```
(defmethod fill :default
  [node value]
  (assoc node :content [(str value)]))

(fill {:span :input} "hello")
;= {:content ["hello"], :span :input}
(fill {:span :input} "hello")
;= {:content ["hello"], :span :input}
```

Этот прием не только работает, но и позволяет даже избавиться от реализации для значения `:div`, поскольку она совпадает с реализацией по умолчанию!

Однако наши значения выбора уже являются ключевыми словами. Это означает, что можно столкнуться с проблемами, если потребуется добавить обработку тега <default>[1].

К нашей радости, defmulti принимает параметры, позволяющие определить, какое значение выбора будет играть роль значения по умолчанию:

```
(defmulti fill
  "Заполняет узел xml/html (согласно модели clojure.xml)
   указанным значением."
  (fn [node value] (:tag node))
  :default nil)                                          ❶

(defmethod fill nil                                      ❷
  [node value]
  (assoc node :content [(str value)]))

(defmethod fill :input
  [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill :default                                 ❸
  [node value]
  (assoc-in node [:attrs :name] (str value)))
```

❶ Параметры метода defmulti – это простые пары ключ/значение. Здесь значением выбора по умолчанию назначается значение nil.
❷ Соответствующая реализация по умолчанию.
❸ Эта реализация более не является реализацией по умолчанию – она обрабатывает элементы <default>.

Навстречу иерархиям

Давайте переделаем fill так, чтобы обеспечить возможность применения различных реализаций к различным тегам <input>. Например, было бы желательно, чтобы радиокнопки (radio buttons) и флажки (checkboxes) помечались, при совпадении их атрибутов value со значениями аргументов, передаваемых мультиметоду fill.

---

[1] Если предположить, что речь идет об обработке некоторого специализированного формата XML или о формате HTML5.

В настоящий момент функция выбора не возвращает информацию об атрибутах, поэтому, чтобы обеспечить возможность выбора по атрибуту `type`, необходимо изменить функцию выбора:

```
(ns-unmap *ns* 'fill)

(defn- fill-dispatch [node value]
  (if (= :input (:tag node))
    [(:tag node) (-> node :attrs :type)]
    (:tag node)))

(defmulti fill
  "Заполняет узел xml/html (согласно модели clojure.xml)
   указанным значением."
  #'fill-dispatch                                    ❶
  :default nil)

(defmethod fill nil
  [node value]
  (assoc node :content [(str value)]))

(defmethod fill [:input nil]
  [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [:input "hidden"]
  [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [:input "text"]
  [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [:input "radio"]
  [node value]
  (if (= value (-> node :attrs :value))
    (assoc-in node [:attrs :checked] "checked")
    (update-in node [:attrs] dissoc :checked)))

(defmethod fill [:input "checkbox"]
  [node value]
  (if (= value (-> node :attrs :value))
    (assoc-in node [:attrs :checked] "checked")
```

```
    (update-in node [:attrs] dissoc :checked)))

(defmethod fill :default
  [node value]
  (assoc-in node [:attrs :name] (str value)))
```

❶ Используя `#'fill-dispatch` вместо `fill-dispatch` мы добавили еще один уровень косвенности, позволяющий изменять функцию выбора, не затрагивая `ns-unmap` и не теряя прежде определенные методы. Вызов `fill-dispatch` фиксирует значение функции выбора, получившееся к моменту, когда форма `defmulti` уже была выполнена, и функция выбора не будет обновляться в дальнейшем. Это очень удобно при разработке кода в REPL.

> **Внимание! Переопределение мультиметода не влечет за собой изменение функции выбора.** Обратите внимание, что в примере выше мы применили `ns-unmap` к `fill`, исключив ее из нашего пространства имен, поэтому мы можем переопределить ее. При необходимости переопределить обычную функцию делать это необязательно, но форма `defmulti` имеет семантику `defonce`, поэтому функции выбора невозможно изменить, не удалив предварительно корневую переменную мультиметода. Это означает, что, что вы должны удалить ее из текущего пространства имен перед переопределением, иначе внесенные вами изменения будут проигнорированы!

Одна интересная особенность этой итерации заключается в том, что значение выбора теперь может быть значением `nil`, ключевым словом или парой ключ/строка. *Значения выбора могут быть не только ключевыми словами*. Однако вскоре мы увидим, что ключевые слова все еще играют важную роль в мультиметодах.

А теперь проверим новый код:

```
(fill {:tag :input
       :attrs {:value "first choice"
                 :type "checkbox"}}
       "first choice")                          ❶
;= {:tag :input,
;=  :attrs {:checked "checked",
;=            :type "checkbox",
;=            :value "first choice"}}
(fill *1 "off")                                 ❷
;= {:tag :input
;=  :attrs {:type "checkbox",
;=            :value "first choice"}}
```

❶ При передаче мультиметоду `fill` флажка с соответствующим значением атрибута `:value`, возвращается отмеченный флажок...

❷ ...а при передаче отмеченного флажка с любым другим значением атрибута `:value`, возвращается неотмеченный флажок[1].

Результат получился вполне удовлетворительный, но в последней итерации было добавлено слишком много повторяющегося кода: для обработки флажков и радиокнопок можно было бы использовать общую реализацию, как и для текстовых полей, явных и неявных (`nil`). Кроме того, реализация для обработки текстовых полей должна использоваться по умолчанию и применяться к элементам `<input>` неизвестного типа.

Проще говоря, флажки и радиокнопки должны обрабатываться как помечаемые поля ввода (checkable inputs), а поля ввода неизвестного типа должны обрабатываться как текстовые поля ввода. Реализовать это можно, определив иерархию, позволяющую выразить отношения между значениями выбора, которая будет использоваться нашим мультиметодом для выбора той или ной реализации.

Иерархии

Мультиметоды в языке Clojure позволяют определять иерархии для поддержки требований, касающихся отношений, включая множественное наследование. Такие иерархии определяются в терминах отношений между именованными объектами[2] (ключевыми словами или символами[3]) и классами.

Здесь не случайно использован термин «иерархии» в множественном числе: имеется возможность создать более одной иерархии. С помощью `make-hierarchy` можно определить одну глобальную иерархию (она же – иерархия по умолчанию) и множество дополнительных иерархий. Кроме того, иерархии и мультиметоды не ограничены единственным пространством имен: вы можете распространять иерархии (посредством `derive`) и мультиметоды (посредством `defmethod`) на любые пространства имен, а не только на те, где они определяются.

---

[1] `*1` представляет значение последнего выражения, вычисленного в интерактивной оболочке REPL. См. раздел «Переменные, создаваемые оболочкой REPL» в главе 10.

[2] Именованные объекты – это объекты, к которым можно применить функции `name` и `namespace`. Они реализуют интерфейс `clojure.lang.Named`.

[3] Обычно ключевые слова являются более предпочтительными.

Глобальная иерархия является совместно используемой, поэтому доступ к ней является более ограниченным. А именно, *в глобальной иерархии не могут использоваться ключевые слова (и символы), не привязанные к пространству имен*. Это защищает от возможных конфликтов две библиотеки, которые могли бы выбрать одинаковые ключевые слова для представления различных семантик.

Отношения в иерархии определяются с помощью `derive`:

```
(derive ::checkbox ::checkable)     ❶
;= nil
(derive ::radio ::checkable)
;= nil
(derive ::checkable ::input)
;= nil
(derive ::text ::input)
;= nil
```

❶ Напомню, что форма записи `::keyword` является сокращенным вариантом формы записи `:current.namespace/keyword`; то есть, `::checkbox` здесь является эквивалентом `:user/checkbox`. Не забывайте также, что `::keyword` соотносится с `:keyword` так же, как `'symbol` соотносится с `'symbol`. Подробнее об этом рассказывается в разделе Ключи (keywords)», в главе 1.

Мы только что описали отношения между различными «классами»: флажки (checkboxes) и радиокнопки(radio buttons) относятся к классу «помечаемых» (checkable) элементов, а все «помечаемые» и «текстовые» (text) элементы относятся к элементам ввода (input). Проверить эти отношения внутри иерархии можно с помощью `isa?`:

```
(isa? ::radio ::input)     ❶
;= true
(isa? ::radio ::text)
;= false
```

❶ Наследование является транзитивным: класс `::radio` является производным от `::checkable`, который в свою очередь является производным от `::input`.

Внимание. Функция `isa?` редко используется где-либо, кроме REPL. Если она слишком часто применяется в коде, это может означать, что в данном случае можно использовать мультиметод. Это очень напоминает ситуацию с функцией `instance?`: обычно ее присутствие в коде указывает на

возможность использовать некоторый механизм диспетчеризации (интерфейс Java, протокол или мультиметод).

Существует еще несколько функций интроспекции: `underive`, `ancestors`, `parents` и `descendants`, которые удобно использовать при работе в оболочке REPL или для выполнения некоторых трюков, связанных с метапрограммированием.

Классы и интерфейсы тоже могут участвовать в иерархиях, но *только в качестве потомков* и никогда в качестве родителей[1]. Иначе говоря, за пределами иерархии классов, неявно определяемой списком путей classpath в настройках Clojure[2], классы и интерфейсы, могут быть только листьями в дереве иерархии.

```
(isa? java.util.ArrayList Object)
;= true
(isa? java.util.ArrayList java.util.List)
;= true
(isa? java.util.ArrayList java.util.Map)        ❶
;= false
(derive java.util.Map ::collection)             ❷
;= nil
(derive java.util.Collection ::collection)
;= nil
(isa? java.util.ArrayList ::collection)         ❸
;= true
(isa? java.util.HashMap ::collection)
;= true
```

❶ В инфраструктуре Java Collections Framework, классы `Map` и `Collection` не связаны родственными отношениями. Поэтому отсутствует механизм выбора, который позволил бы, опираясь исключительно на эти статические типы, обеспечить возможность обработки `Map` и `Collection` в единственном методе (за исключением отступления, например, к типу `Object`).

❷ Мы можем объявить, что `Map` и `Collection` являются производными от нового идентификатора `::collection` в глобальной иерархии.

[1] Единственный способ создать производную от класса или интерфейса – определить тип с использованием формы взаимодействия или с помощью `deftype`, `defrecord` или `reify`.

[2] Дополнительную информацию о списке путей classpath можно найти в разделе «Знакомство с classpath», в главе 8.

❸ Теперь ключ `::collection` можно использовать в качестве значения выбора в определении `defmethod`, так как теперь ему будут соответствовать любые классы, реализующие `Map` или `Collection`.

Однако этот аспект иерархий не имеет отношения к нашему примеру. Мы еще вернемся к теме использования классов и интерфейсов в иерархиях в разделе «Множественное наследование» ниже.

Примечание. Иерархия классов Java всегда является частью любой иерархии, даже вновь созданной с помощью `make-hierarchy`.

```
(def h (make-hierarchy))
;= #'user/h
(isa? h java.util.ArrayList java.util.Collection)
;= true
```

Таким образом, функция `isa?` является расширенной версией `instance?`, в том смысле, что с ее помощью можно проверить, является ли один класс производной от другого класса, или реализует ли класс некоторый интерфейс.

Независимые иерархии

В настоящий момент `fill-dispatch` может возвращать `nil`, ключевое слово или вектор. Значения `nil` и векторы не могут участвовать в иерархиях, а ключи не могут участвовать в глобальной иерархии, потому что возвращаемые ключевые слова не содержат в себе пространство имен.

Итак, у нас на выбор остается два варианта: либо возвращать из `fill-dispatch` ключевые слова, содержащие имя пространства имен, либо использовать приватную иерархию.

`derive` неявно изменяет глобальную иерархию, но при использовании собственной иерархии вы должны будете сами управлять изменениями. Это легко организовать, если сохранить иерархию в ссылочном типе, таком как ссылка, атом или переменная. Переменная представляется вполне безопасным выбором[1]: так как для хранения глобальной иерархии так же используется переменная.

---

[1] Потому что иерархии обычно изменяются нечасто, и было бы желательно, чтобы изменения в них были видимы во всех потоках выполнения. При наличии в программе других требований (таких как применение транзакций или динамических областей видимости) выберите другой, более подходящий ссылочный тип. Подробности смотрите в главе 4.

Пример 7.1. Реализация fill с использованием собственной иерархии

```
(ns-unmap *ns* 'fill)

(def fill-hierarchy (-> (make-hierarchy)                          ❶
                        (derive :input.radio ::checkable)
                        (derive :input.checkbox ::checkable)
                        (derive ::checkable :input)
                        (derive :input.text :input)
                        (derive :input.hidden :input)))

(defn- fill-dispatch [node value]
  (if-let [type (and (= :input (:tag node))
                     (-> node :attrs :type))]
    (keyword (str "input." type))
    (:tag node)))

(defmulti fill
  "Заполняет узел xml/html (согласно модели clojure.xml)
   указанным значением."
  #'fill-dispatch
  :default nil
  :hierarchy #'fill-hierarchy)                                    ❷

(defmethod fill nil [node value]
  (assoc node :content [(str value)]))

(defmethod fill :input [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill ::checkable [node value]
  (if (= value (-> node :attrs :value))
    (assoc-in node [:attrs :checked] "checked")
    (update-in node [:attrs] dissoc :checked)))
```

❶ Использование пары `input.type` и `::checkable` позволяет избежать возможных конфликтов с допустимыми именами тегов. Точка (`.`) в ключах не имеет специального значения – это лишь часть имени.

❷ В параметре `:hierarchy` мультиметоды ожидают получить ссылку того или иного вида на иерархию, а не саму иерархию. Благодаря этому во время выполнения можно при необходимости изменять иерархию. В данном случае мы передаем *переменную* (var) `fill-hierarchy` (а не ее значение, которое было получено в момент вычисления формы `defmulti`) и можем использовать `alter-var-root` для изменения иерархии динамически, не переопределяя корень мультиметода.

> **Внимание.** Все функции для работы с иерархиями (такие как `derive`, `isa?`, `parents` и другие) могут принимать значение иерархии в первом дополнительном аргументе. Из этих функций особенно выделяется `derive`, которая без аргумента со значением иерархии производит побочный эффект – модифицирует глобальную иерархию – а при наличии такого аргумента является чистой функцией!

На данном этапе разработки мультиметода `fill` нам необходимо было избавиться от повторяющегося кода, и теперь эта цель достигнута. Однако вторая задача не была решена: элементы ввода неизвестного типа, такие как текстовые поля ввода, пока не обрабатываются.

```
(fill {:tag :input
       :attrs {:type "date"}}
      "20110820")
;= {:content ["20110820"], :attrs {:type "date"}, :tag :input}
```

Не совсем то, что мы имели в виду! Мы не указали, что элементы ввода неизвестного типа должны интерпретироваться как текстовые. Проблема в том, что множество элементов ввода неизвестных типов остается открытым, поэтому мы не можем определить все разновидности. И так как множество всех значений выбора больше, чем множество значений выбора, описывающих теги элементов ввода, мы не можем использовать значение по умолчанию для обработки всех неизвестных типов элементов ввода.

Или можем? Если мыслить *динамически*, избавившись от предубеждения, что мы должны *статически* определить иерархию заранее, можно заметить, что иерархии *не* статичны. То есть, наше значение по умолчанию может действовать как страховка и динамически определять необходимое отношение так, чтобы однозначно соответствовать каждому новому типу элемента ввода.

Пример 7.2. Динамическое изменение иерархии, используемой мультиметодом fill

```
(defmethod fill nil [node value]
  (if (= :input (:tag node))
    (do
      (alter-var-root #'fill-hierarchy
        derive (fill-dispatch node value) :input)    ❶
      (fill node value))                             ❷
    (assoc node :content [(str value)])))
```

❶ Мы изменили значение переменной динамически, указав, что значение выбора для узла :input неизвестного типа является производным от :input в нашей иерархии.
❷ После внесения этого изменения мы рекурсивно вызываем fill, которая теперь, после изменения иерархии, передаст управление реализации метода :input, подходящей для данного случая.

Этот трюк удался:

```
(fill {:tag :input
       :attrs {:type "date"}}
      "20110820")
;= {:attrs {:value "20110820", :type "date"}, :tag :input}
```

Менее удачное решение заключается в создании дополнительного мультиметода fill-input и вызова его из реализации :input мультиметода fill.

```
(ns-unmap *ns* 'fill)

(def input-hierarchy (-> (make-hierarchy)
                         (derive :input.radio ::checkable)
                         (derive :input.checkbox ::checkable)))    ❶

(defn- fill-dispatch [node value]
  (:tag node))

(defmulti fill
  "Заполняет узел xml/html (согласно модели clojure.xml)
   указанным значением."
  #'fill-dispatch
  :default nil)                                                    ❷

(defmulti fill-input
  "Заполняет поле ввода."
  (fn [node value] (-> node :attrs :type))
  :default nil
  :hierarchy #'input-hierarchy)

(defmethod fill nil [node value]
  (assoc node :content [(str value)]))

(defmethod fill :input [node value]
```

```
  (fill-input node value))

(defmethod fill-input nil [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill-input ::checkable [node value]
  (if (= value (-> node :attrs :value))
    (assoc-in node [:attrs :checked] "checked")
    (update-in node [:attrs] dissoc :checked)))
```

❶ Больше не нужно явно определять принадлежность `:text` и `:hidden` к случаю по умолчанию. К тому же это невозможно, потому что значением выбора по умолчанию является `nil`, которое не может участвовать в иерархиях.

❷ `fill` больше не зависит от нашей иерархии – только `fill-input`.

Сделаем выбор по-настоящему множественным!

До сих пор в нашем примере осуществлялся выбор реализации без учета типа, но это не был множественный выбор: второй аргумент (`value`) игнорировался во всех предыдущих функциях выбора.

В множественном выборе нет ничего сложного, потому что в большинстве случаев он осуществляется точно так же: функция выбора вычисляет одно значение, на основе которого выбирается реализация, соответствующая используемой иерархии. Система поддержки мультиметодов не знает, что наши функции выбора принимают во внимание только первый аргумент.

Однако значения выбора, являющиеся векторами, обрабатываются функцией `isa?` поэлементно[1]:

```
(isa? fill-hierarchy [:input.checkbox :text] [::checkable :input])
;= true
```

Как уже отмечалось, иерархия классов Java включается во все иерархии, а это значит, что при необходимости можно добавлять в смесь некоторые классы:

```
(isa? fill-hierarchy [:input.checkbox String] [::checkable CharSequence])
;= true
```

[1] Она выполняется рекурсивно: в качестве значений выбора можно даже использовать векторы векторов!

Мы воспользуемся этой возможностью, чтобы сделать мультиметод `fill` более интеллектуальным и учитывать тип его аргумента `value`.

Для начала изменим функцию `fill-dispatch` так, чтобы она возвращала вектор с одним ключевым словом и с одним классом[1].

```
(defn- fill-dispatch [node value]
  (if-let [type (and (= :input (:tag node))
                     (-> node :attrs :type))]
    [(keyword (str "input." type)) (class value)]
    [(:tag node) (class value)]))
```

Теперь нам хотелось бы, чтобы в базовом варианте значение преобразовывалось в строку, независимо от его типа. Однако флажки принимают множества в виде значений: флажок будет помечен, только если его значение присутствует в множестве.

```
(ns-unmap *ns* 'fill)

(def fill-hierarchy (-> (make-hierarchy)
                        (derive :input.radio ::checkable)
                        (derive :input.checkbox ::checkable)))

(defn- fill-dispatch [node value]
  (if-let [type (and (= :input (:tag node))
                     (-> node :attrs :type))]
    [(keyword (str "input." type)) (class value)]
    [(:tag node) (class value)]))

(defmulti fill
  "Заполняет узел xml/html (согласно модели clojure.xml)
   указанным значением."
  #'fill-dispatch
  :default nil
  :hierarchy #'fill-hierarchy)

(defmethod fill nil
  [node value]
  (if (= :input (:tag node))
    (do
      (alter-var-root #'fill-hierarchy
        derive (first (fill-dispatch node value)) :input)     ❶
```

[1] Этот пример основан на примере 7.1 и использует прием динамической модификации иерархий, демонстрировавшийся в примере 7.2.

```
       (fill node value))
     (assoc node :content [(str value)])))

(defmethod fill
  [:input Object] [node value]
  (assoc-in node [:attrs :value] (str value)))

(defmethod fill [::checkable clojure.lang.IPersistentSet]
  [node value]
  (if (contains? value (-> node :attrs :value))
    (assoc-in node [:attrs :checked] "checked")
    (update-in node [:attrs] dissoc :checked)))
```

❶ Здесь функция `first` используется, чтобы оставить только ключевое слово из значения выбора. Напомню: в иерархиях можно использовать только ключевые слова, символы или классы.

Теперь можно помечать и сбрасывать флажки, используя более гибкую форму записи множеств:

```
(fill {:tag :input
       :attrs {:value "yes"
               :type "checkbox"}}
           #{"yes" "y"})
;={:attrs {:checked "checked", :type "checkbox", :value "yes"}, :tag :input}
(fill *1 #{"no" "n"})
;= {:attrs {:type "checkbox", :value "yes"}, :tag :input}
```

При этом другие элементы ввода, а также элементы, не являющиеся элементами ввода, будут заполняться, как и предусматривалось:

```
(fill {:tag :input :attrs {:type "text"}} "some text")
;= {:attrs {:value "some text", :type "text"}, :tag :input}
(fill {:tag :h1} "Big Title!")
;= {:content ["Big Title!"], :tag :h1}
```

Кое-что еще

Множественное наследование

Наш действующий пример мультиметода `fill` использует слишком простую иерархию, недостаточную для ввода множественного «наследования». Такие отношения часто возникают в мультиметодах, которые имеют дело с интерфейсами.

Допустим, нам необходимо реализовать функцию `run`, которая может запускать нечто, способное запускаться (например, реализующее интерфейсы `java.lang.Runnable` и `java.util.concurrent.Callable`):

```
(defmulti run "Запускает вычисления." class)

(defmethod run Runnable
  [x]
  (.run x))

(defmethod run java.util.concurrent.Callable
  [x]
  (.call x))
```

Протестируем с помощью функции:

```
(run #(println "hello!"))
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   Multiple methods in multimethod 'run' match dispatch value:
;=     class user$fn__1422 -> interface java.util.concurrent.Callable and
;=       interface java.lang.Runnable, and neither is preferred>
```

Исключение сопровождается достаточно информативным сообщением об ошибке: так как функции в языке Clojure реализуют оба интерфейса, `Runnable` и `Callable`, мультиметод не может решить, какую реализацию выбрать и подсказывает, что какой-то одной должно быть отдано *предпочтение* (preferred).

Предпочтения определяются с помощью функции `prefer-method`. Эта функция принимает три аргумента: мультиметод, для которого определяются предпочтения, и *два* значения выбора, первое из которых будет считаться более предпочтительным перед вторым:

```
(prefer-method run java.util.concurrent.Callable Runnable)
;= #<MultiFn clojure.lang.MultiFn@6dc98c1b>
(run #(println "hello!"))
;= hello!
;= nil
```

Теперь мультиметод знает, какой реализации отдать предпочтение, и выполняет запуск без каких-либо проблем.

Данный механизм предпочтений позволяет декларативно разрешать проблемы *ромбовидного наследования*[1], то есть ситуации, когда один класс наследует один и тот же суперкласс через два или более промежуточных суперкласса. Это позволяет проектировать иерархии без страха перед множественным наследованием. *Механизм предпочтений делает множественное наследование явным и потому более простым в применении.*

Интроспекция мультиметодов

Существует несколько редко используемых функций, позволяющих применять приемы метапрограммирования при работе с мультиметодами: `remove-method`, `remove-all-methods`, `prefers`, `methods` и `get-method`. Эти функции дают возможность получать и изменять мультиметоды.

Обратите внимание, однако, на отсутствие функции `add-method`. Помните, что макрос `defmethod`, не смотря на то, что является формой определения, не обязательно должен быть выражением верхнего уровня. Однако иногда может потребоваться зарегистрировать в качестве реализации метода существующую функцию. Как это сделать, можно узнать, заглянув за кулисы:

```
(macroexpand-1 '(defmethod mmethod-name dispatch-value [args] body))
;= (. mmethod-name clojure.core/addMethod dispatch-value (clojure.core/fn
    [args] body))    ❶
```

❶ Вывод `clojure.core/addMethod` вместо простого имени `addMethod` является результатом действия `syntax-quote`. Предотвратить это сложно, но Clojure достаточно интеллектуален, чтобы игнорировать пространства имен в именах методов Java.

Исходя из этого, сама собой напрашивается простая реализация функции `add-method`:

```
(defn add-method [multifn dispatch-val f]
  (.addMethod multifn dispatch-val f))
```

[1] Название «ромбовидное» не всегда верно, так как иерархии в Clojure не имеют универсального корня. В Clojure эту проблему точнее было бы назвать проблемой V-образного наследования.

Так получилось, что эта функция уже присутствует в пространстве имен `clojure.pprint` в виде приватной функции под именем `use-method`. Однако при использовании любой функции, `use-method` или представленной выше `add-method`, вы попадаете в зависимость от особенностей реализации, поэтому готовьтесь потратить дополнительные усилия на их сопровождение в будущих версиях Clojure.

type и class; или месть ассоциативного массива

Форма `class` имеет родственную ей форму `type`. Обычно выражение `(type x)` возвращает тот же результат, что и выражение `(class x)`, за исключением случаев, когда `x` имеет слот `:type` в метаданных:

```
(class {})
;= clojure.lang.PersistentArrayMap
(type {})
;= clojure.lang.PersistentArrayMap
(class ^{:type :a-tag} {})
;= clojure.lang.PersistentArrayMap
(type ^{:type :a-tag} {})
;= :a-tag
```

Слот `:type` в метаданных позволяет без лишних церемоний указывать типы данных и использовать эти типы в мультиметодах. Обратите внимание, что данный прием работает также при добавлении метаданных к другим разновидностям объектов: векторам, множествам, функциям и так далее.

Например, расширим пример из раздела «Множественное наследование» так, чтобы мультиметод `run` принимал реализации интерфейсов `Runnable`, `Callable`, обычные функции Clojure, а также ассоциативные массивы, «типизированные» как запускаемые объекты и содержащие что-то из вышеперечисленного в слоте `:run`:

```
(ns-unmap *ns* 'run)

(defmulti run "Запускает вычисления." type)      ❶

(defmethod run Runnable
  [x]
  (.run x))

(defmethod run java.util.concurrent.Callable
```

```
  [x]
  (.call x))

(prefer-method run java.util.concurrent.Callable Runnable)

(defmethod run :runnable-map
  [m]
  (run (:run m)))

(run #(println "hello!"))
;= hello!
;= nil
(run (reify Runnable
       (run [this] (println "hello!"))))
;= hello!
;= nil
(run ^{:type :runnable-map}
     {:run #(println "hello!") :other :data})
;= hello!
;= nil
```

❶ Теперь форма `type` действует как функция выбора. Она будет возвращать значение слота `:type` из метаданных любого ассоциативного массива или класс аргумента, если этот слот в метаданных отсутствует.

Конечно, того же результата можно добиться реализовав в функции выбора явную проверку наличия слота `:run` в любом ассоциативном массиве. Однако представьте, что в какой-то момент вам потребуется организовать возможность вызова функции, находящейся в последнем элементе вектора, созданного где-то в приложении. Для этого пришлось бы изменить функцию выбора, тем самым дискредитировав идею мультиметодов (и многих других возможностей языка Clojure), как механизма, позволяющего извлекать операции из данных, участвующих в операциях.

Функции выбора не имеют ограничений

В нашем заключительном примере в этой главе мы продемонстрируем большую гибкость, присущую мультиметодам. До сих пор мы создавали мультиметоды, функции выбора которых возвращают значения, основанные исключительно на их аргументах. Однако это не является обязательным требованием.

Рассмотрим систему обмена сообщениями, где сообщения обрабатываются по-разному, в зависимости от их приоритетов:

```
(def priorities (atom {:911-call :high
                       :evacuation :high
                       :pothole-report :low
                       :tree-down :low}))

(defmulti route-message
  (fn [message] (@priorities (:type message))))

(defmethod route-message :low
  [{:keys [type]}]
  (println (format "Oh, there's another %s. Put it in the log." (name type))))

(defmethod route-message :high
  [{:keys [type]}]
  (println (format "Alert the authorities, there's a %s!" (name type))))
```

Эта реализация выглядит достаточно просто[1]:

```
(route-message {:type :911-call})
;= Alert the authorities, there's a 911-call!
;= nil
(route-message {:type :tree-down})
;= Oh, there's another tree-down. Put it in the log.
;= nil
```

Однако как быть, если приоритеты сообщений могут изменяться динамически? Никаких проблем: просто адаптируйте данные, управляющие функцией выбора, и поведение `route-message` будет изменяться автоматически, *без изменения кода или данных*:

```
(swap! priorities assoc :tree-down :high)
;= {:911-call :high, :pothole-report :low, :tree-down :high,
;=  :evacuation :high}
(route-message {:type :tree-down})
;= Alert the authorities, there's a tree-down!
;= nil
```

[1] Наши реализации `route-message` определенно могли бы делать что-то большее, чем просто выводить сообщения в поток стандартного вывода.

Это открывает возможности, достаточно широкие для решения задач, стоящих перед вами. В то же время – извините за оксюморон – этих возможностей может оказаться достаточно, чтобы запутаться в них. Мультиметод, чье поведение не зависит от аргументов, по определению является не идемпотентным[1]. Функции не становятся от этого менее надежными или менее полезными, они просто становятся более сложными для понимания, тестирования и композиции с другими функциями, в сравнении с их идемпотентными сестрами.

В заключение

На примере разработки `fill` мы рассмотрели все основные особенности мультиметодов в языке Clojure[2].

Должно пройти какое-то время, прежде чем вы освоитесь с широтой возможностей мультиметодов, научитесь мыслить не только категориями выбора на основе типов. Каждый раз, обнаруживая, что вы пишете множество вложенных условий, или длинную форму `cond`, или определяете массу типов, только чтобы связать свои данные с определенной функциональностью, вы должны задать себе вопрос – а не лучше ли будет применить мультиметод.

---

[1] Подробнее об идемпотентности, чистых функциях и их преимуществах рассказывается в разделе «Чистые функции», в главе 2.

[2] Дополнительные примеры мультиметодов можно найти в главе 15.

Часть III
ИНСТРУМЕНТЫ, ПЛАТФОРМЫ И ПРОЕКТЫ

Глава 8. Создание и организация проектов на Clojure

Как ни странно, самые важные аспекты, побуждающие приступить к использованию нового, многообещающего языка программирования, часто никак не связаны с самим языком: вам необходимо организовать свой программный код на новом языке в компоненты, которые могут распространяться и использоваться, либо другими программистами (в виде библиотек) и конечными пользователями, либо устанавливаться, например, на сервер (в виде веб-приложений). Специфические особенности этой задачи могут существенно изменяться в зависимости от того, используете ли вы новый язык для разработки части существующего проекта или совершенно нового проекта, а также от конкретных требований на этапе развертывания.

Нельзя в одной главе охватить все возможные способы организации проектов и описать их достоинства. Расхождения во мнениях часто могут возобладать над важностью обсуждаемой темы[1], однако выбор пути, соответствующего типичным подходам, принятым в сообществе Clojure, зависит исключительно от вас. В этой главе мы дадим вам ряд общих рекомендаций, касающихся организации программного кода, и подскажем, как лучше решать проблемы, связанные со сборкой проектов на языке Clojure, используя два инструмента, Leiningen и Maven, наиболее популярных в сообществе Clojure.

География проекта

Прежде чем погружаться в механику сборки проектов, сначала необходимо определиться с организационными моментами, касающимися физического размещения файлов, а также функциональной

---

[1] Вокруг «организации» проектов разгораются, пожалуй, еще более жаркие споры, чем такие «дебаты», как «табуляции против пробелов» и «emacs против vi»: http://bikeshed.org.

организации программного кода. Это означает, что мы должны поговорить о *пространствах имен*.

Определение и использование пространств имен

Как говорилось в разделе «Пространства имен», в главе 1[1], пространства имен в языке Clojure:

- динамически связывают символы с именами Java-классов и переменными, последние из которых могут содержать любые указанные вами значения (чаще всего функции, константы и значения ссылочных типов);
- являются примерными аналогами пакетов в Java и модулей в Python и Ruby.

Весь программный код на языке Clojure находится внутри пространств имен. Если вы не определите собственное пространство имен, все переменные будут помещаться в пространство имен по умолчанию `user`. Оно отлично подходит для экспериментов в оболочке REPL, но использовать его для организации кода и данных, которыми будут пользоваться другие, – не самая лучшая идея. Мы должны знать идиоматические приемы определения пространств имен, как они связаны с отдельными исходными файлами и как лучше использовать их для организации и структурирования программного кода. В Clojure имеются отдельные функции для управления мельчайшими особенностями пространств имен (их очень удобно использовать в REPL), объединенные также под покровом единственного макроса, который можно использовать для объявления в одном месте имени пространства имен, размещения верхнеуровневой документации и описания зависимостей от других пространств имен и Java-классов.

in-ns. Форма `def` и все ее варианты (такие как `defn`) определяют переменные внутри текущего пространства имен, на которое всегда ссылается символ `*ns*`:

```
*ns*
;= #<Namespace user>
```

[1] Если вы еще не читали этот раздел, сделайте это прямо сейчас. В нем даются самые основные понятия, касающиеся пространств имен, рассказывается о символах и переменных, и как первые разрешаются во вторые.

```
(defn a [] 42)
;= #'user/a
```

С помощью `in-ns` можно переключаться между пространствами имен (попутно создавая несуществующие), тем самым обеспечивая возможность определения переменных в других пространствах имен:

```
(in-ns 'physics.constants)
;= #<Namespace physics.constants>
(def ^:const planck 6.62606957e-34)
;= #'physics.constants/planck
```

Однако вы скоро обнаружите, что в новом пространстве имен что-то не так:

```
(+ 1 1)
;= #<CompilerException java.lang.RuntimeException:
;=    Unable to resolve symbol: + in this context,
;=    compiling:(NO_SOURCE_PATH:1)>
```

Функция `+` (и все остальные функции из пространства имен `clojure.core`) оказались недоступны, так как они находятся в пространстве имен по умолчанию `user`, в котором мы работали все время. Однако к ним можно получить доступ, используя символы, квалифицированные пространством имен:

```
(clojure.core/range -20 20 4)
;= (-20 -16 -12 -8 -4 0 4 8 12 16)
```

Не забывайте, что пространства имен связывают символы с переменными, а `in-ns` переключается в указанное пространство имен, это все, что делает данная функция. Специальные формы в новом пространстве имен (включая `def`, `var`, `.` и другие) все еще остаются доступными, но чтобы использовать более удобную и краткую форму записи, программный код из других пространств имен необходимо загрузить в текущее.

refer. Если предположить, что некоторое пространство имен уже загружено, мы можем использовать `refer` для добавления связей между символами и переменными из него в наше пространство имен. Выше мы объявили функцию `a` в пространстве имен `user`. Чтобы упростить доступ к функции `a`, можно скопировать в текущее пустое пространство имен все общедоступные связи между символами и переменными из пространства имен `user`:

```
user/a
;= #<user$a user$a@6080669d>
(clojure.core/refer 'user)
;= nil
(a)
;= 42
```

Теперь символ `a` в текущем пространстве имен связан с переменной `user/a`, и мы можем использовать ее как если бы она была определена локально. Это определенно проще, чем писать символы, квалифицированные пространствами имен, для обращения к переменным в других пространствах имен.

Однако функция `refer` имеет более широкую область применения, чем простое «импортирование»: она позволяет исключать, включать и переименовывать отдельные переменные при отображении в текущее пространство имен, используя необязательные именованные аргументы `:exclude`, `:only` и `:rename`, соответственно. Например, применим `refer` к пространству имен `clojure.core`, исключив некоторые функции и определив другие локальные имена для некоторых арифметических операторов:

```
(clojure.core/refer 'clojure.core
  :exclude '(range)
  :rename '{+ add
            - sub
            / div
            * mul})
;= nil
(-> 5 (add 18) (mul 2) (sub 6))
;= 40
(range -20 20 4)
;= #<CompilerException java.lang.RuntimeException:
;=   Unable to resolve symbol: range in this context,
;=   compiling:(NO_SOURCE_PATH:1)>
```

Теперь мы можем использовать все общедоступные функции[1] из `clojure.core` (кроме `range`, которая была исключена) и употреблять некоторые арифметические функции под другими именами.

---

[1] `refer` не копирует ссылки на приватные переменные из исходного пространства имен. Подробнее о приватных переменных рассказывается в разделе «Переменные», в главе 4.

Пространство имен `clojure.core` всегда загружается предварительно (и импортируется в пространство имен `user`), однако зачастую этого бывает недостаточно и необходимо определить множество собственных пространств имен, чтобы структурировать свой программный код. Для этого необходим инструмент загрузки пространств имен.

> **Примечание.** Функция `refer` редко используется непосредственно, но косвенно ее возможности доступны через функцию `use`, используемую более широко.

require и use. Когда возникает потребность в общедоступных функциях или данных из другого пространства имен, применяются функции `require` и `use`, которые:

1. Гарантируют загрузку указанных пространств имен.
2. При необходимости определяют псевдонимы для символов из этих пространств имен.
3. Неявно вызывают `refer`, чтобы обеспечить возможность ссылаться на переменные из других пространств имен без использования префиксов.

Функция `require` обеспечивает пункты (1) и (2); функция `use`, основанная на `require` и `refer`, обеспечивает поддержку пункта (3).

Давайте запустим новый сеанс REPL и попробуем воспользоваться функцией `union` из пространства имен `clojure.set`:

```
(clojure.set/union #{1 2 3} #{4 5 6})
;= #<ClassNotFoundException java.lang.ClassNotFoundException: clojure.set>
```

Стоп! Это пространство имен еще не загружено – предварительно загружается только `clojure.core`. Сначала нужно загрузить пространство имен `clojure.set`, отыскав его в списке путей classpath[1], с помощью `require`, и только потом мы получим возможность использовать функции из него:

```
(require 'clojure.set)
;= nil
(clojure.set/union #{1 2 3} #{4 5 6})
;= #{1 2 3 4 5 6}
```

[1] См. ниже раздел «Пространства имен и файлы», где описываются соответствия между пространствами имен и файлами, и раздел «Знакомство с classpath», где рассказывается, что такое список путей classpath и какую роль он играет.

Однако, необходимость использовать полностью квалифицированные символы для обращения к переменным достаточно утомительна, особенно если используемые вами библиотеки включают пространства имен с длинными названиями или с названиями, состоящими из нескольких сегментов. К счастью, `require` позволяет определять псевдонимы для пространств имен:

```
(require '[clojure.set :as set])    ❶
;= nil
(set/union #{1 2 3} #{4 5 6})
;= #{1 2 3 4 5 6}
```

❶ Вектор аргументов, передаваемый функциям `require` и `use`, иногда называют *libspecs* (спецификация библиотеки): они определяют, как будет загружаться и именоваться библиотека в текущем пространстве имен.

Когда требуется загрузить несколько пространств имен, названия которых начинаются с общего префикса, функции `require` можно передать упорядоченную коллекцию, первым элементом в которой является префикс пространств имен, а остальные содержат дополнительные сегменты в названиях. То есть, если потребуется загрузить два пространства имен, `clojure.set` и `clojure.string`, необязательно повторять префикс `clojure`:

```
(require '(clojure string [set :as set]))
```

Функция `use` обладает всеми возможностями `require`, за исключением того, что по умолчанию вызывает `refer` для указанного пространства имен после загрузки. То есть, выражение `(use 'clojure.xml)` эквивалентно:

```
(require 'clojure.xml)
(refer 'clojure.xml)
```

Кроме того, `use` передает все свои аргументы функции `refer`, благодаря чему можно использовать параметры `:exclude`, `:only` и `:rename` последней. Для иллюстрации рассмотрим ситуацию, когда требуется задействовать пространства имен `clojure.string` и `clojure.set`:

1. Нам хотелось бы скопировать ссылки на все переменные в текущее пространство имен, но...

2. У нас имеется несколько функций, имена которых будут конфликтовать с именами функций в `clojure.string`; определение псевдонима для пространства имен (с помощью `:as` в функции `require`) поможет решить проблему, но...
3. Нам предстоит часто использовать функцию `clojure.string/join`, а это имя не конфликтует ни с одной из функций в текущем пространстве имен, поэтому хотелось бы избежать использования псевдонима пространства имен в данном случае.
4. Оба пространства имен, `clojure.string` и `clojure.set`, определяют функцию `join`; попытка скопировать ссылки на обе эти функции вызовет ошибку, однако нам нужна только функция `clojure.string/join`.

С помощью `use` легко можно удовлетворить эти требования:

```
(use '(clojure [string :only (join) :as str]
               [set :exclude (join)]))
;= nil
join
;= #<string$join clojure.string$join@2259a735>
intersection
;= #<set$intersection clojure.set$intersection@2f7fc44f>
str/trim
;= #<string$trim clojure.string$trim@283aa791>
```

Теперь функцию `join` из `clojure.string` можно вызывать, не указывая пространство имен. Ссылки на остальные функции в `clojure.set` были скопированы в текущее пространство имен (включая `intersection`), а пространство имен `clojure.string` целиком доступно под псевдонимом `str`.

Эффективное использование require, refer и use

Эти функции вместе поддерживают множество различных параметров, особенно если сравнивать их с такими инструментами, как `import` в Java и `require` в Ruby. Эффективное и идиоматическое их использование может вызывать сложности у начинающих осваивать Clojure.

При использовании `require` всегда желательно указывать псевдонимы для всех пространств имен:

```
(require '(clojure [string :as str]
                   [set :as set]))
```

Это примерно эквивалентно инструкции `import sys, os` в Python. Так как названия пространств имен обычно состоят из нескольких сегментов (в противоположность именам большинства модулей в Python), в Clojure не назначаются псевдонимы по умолчанию загружаемым пространствам имен, но допускается управлять используемыми псевдонимами. Конечно, если название пространства имен достаточно короткое, или переменные из него используются в программе лишь несколько раз, тогда можно вообще не указывать псевдоним в `require`.

Также часто рекомендуется использовать функцию `use`, указывая в ней псевдоним пространства имен и явный список переменных, загружаемых в текущее пространство имен:

```
(use '[clojure.set :as set :only (intersection)])
```

Поскольку в таком виде `use` покрывает все возможности, предоставляемые функциями `require` и `refer`, она позволяет объединить определение всех ссылок в одну форму. Даже там, где используется `require` с определением псевдонимов, эквивалентная форма вызова функции `use` получится не только не длиннее, но и позволит легко добавлять функции с помощью аргумента `:only`.

В любом случае, считается хорошей практикой избегать использования функции `use` без наложения ограничений, то есть, без параметра `:only`, явно определяющего список функций, ссылки на которые должны быть скопированы в текущее пространство имен. Это дает возможность сразу выяснить, какие части пространств имен используются в коде, и избежать неожиданных конфликтов при появлении в библиотеках новых функций, имена которых совпадают с именами ваших локальных функций.

import. Основная роль пространств имен в языке Clojure – обеспечить связь между символами и переменными, нередко объявленными в других пространствах имен, однако они также обеспечивают связь между символами и классами и интерфейсами Java. Добавлять такие связи в текущее пространство имен можно с помощью `import`.

Функция `import` принимает полные имена импортируемых классов или упорядоченную коллекцию, описывающую пакет и классы. Импортировав класс, вы получаете возможность использовать его «короткое имя» в текущем пространстве имен:

```
(Date.)                                                    ❶
;= #<CompilerException java.lang.IllegalArgumentException:
;= Unable to resolve classname: Date, compiling:(NO_SOURCE_PATH:1)>
```

```
(java.util.Date.)                                                    ❷
;= #<Date Mon Jul 18 12:31:38 EDT 2011>
(import 'java.util.Date 'java.text.SimpleDateFormat)                 ❸
;= java.text.SimpleDateFormat
(.format (SimpleDateFormat. "MM/dd/yyyy") (Date.))                   ❹
;= "07/18/2011"
```

❶ Класс `Date` находится в пакете `java.util`, поэтому попытка использовать короткое имя класса до его импортирования в текущее пространство имен вызывает ошибку.

❷ Ссылаться на классы и интерфейсы Java можно и не импортируя их, но при этом необходимо указывать их полные имена, которые могут быть отталкивающе длинными.

❸ Классы можно импортировать в текущее пространство имен, указав в вызове функции `import` символы с полными их именами.

❹ После импортирования на классы можно ссылаться по их коротким именам.

По умолчанию все классы из пакета `java.lang` всегда импортируются во все пространства имен; например, сослаться на класс `java.lang.String` можно с помощью символа `String`, не импортируя его явно.

Когда требуется импортировать множество классов из одного пакета, функции `import` можно передать коллекцию с префиксами, определяющими пакеты, по аналогии с `require`:

```
(import '(java.util Arrays Collections))
;= java.util.Collections
(->> (iterate inc 0)
  (take 5)
  into-array
  Arrays/asList
  Collections/max)
;= 4
```

Такое случается редко, но вы должны знать, что нельзя импортировать два класса с одинаковыми короткими именами в одно пространство имен:

```
(import 'java.awt.List 'java.util.List)
;= #<IllegalStateException java.lang.IllegalStateException:
;=    List already refers to: class java.awt.List in namespace: user>
```

Эту проблему можно обойти (как и в Java), если наиболее часто используемый класс импортировать в пространство имен, а для ссылки на другой указывать полное его имя.

> **Внимание.** Несмотря на то, что функция `import` в Clojure концептуально похожа на инструкцию `import` в Java, она имеет несколько важных отличий. Во-первых, в отличие от своего аналога в Java она не поддерживает импортирование с шаблонными символами, как, например: `import java.util.*;`. Если потребуется импортировать несколько классов из одного пакета, придется перечислить их все, в списке, начинающемся с префикса, определяющего имя пакета, как было показано выше.
> Во-вторых, если возникнет необходимость сослаться на внутренний класс (такой как `java.lang.Thread.State` или `java.util.Map.Entry`), вы должны будете использовать нотацию, принятую в Java (например, `java.lang.Thread$State` или `java.util.Map$Entry`). Это относится не только к вызову функции `import`, но и к любым ссылкам на внутренние классы.

ns. Все вспомогательные функции для работы с пространствами имен, описанные выше в этом разделе, в основном предназначены для использования в оболочке REPL. Внутри программ для определения пространств имен следует использовать макрос `ns`[1].

Макрос `ns` позволяет декларативно объявить пространство имен вместе с документацией к нему, а также указать, что необходимо загрузить и импортировать для успешной работы. Это очень тонкая обертка вокруг функций `require`, `refer`, `use` и `import`. Следующую груду вызовов вспомогательных функций:

```
(in-ns 'examples.ns)
(clojure.core/refer 'clojure.core :exclude '[next replace remove])
(require '(clojure [string :as string]
                   [set :as set])
         '[clojure.java.shell :as sh])
(use '(clojure zip xml))
(import 'java.util.Date
```

[1] Может показаться заманчивым, просто скопировать в файл *.clj* код, опробованный в оболочке REPL (вместе с формами `in-ns`, `refer` и другими), и закончить на этом. Мы призываем вас – боритесь с этим искушением. Как будет показано в следующем разделе, существуют определенные правила, касающиеся организации программного кода на языке Clojure, и пренебрежение возможностью определения пространств имен с помощью `ns` идет вразрез с этими правилами и лишает вас дополнительных преимуществ.

```
        'java.text.SimpleDateFormat
        '(java.util.concurrent Executors
                               LinkedBlockingQueue))
```

можно заменить эквивалентным объявлением `ns`:

```
(ns examples.ns
  (:refer-clojure :exclude [next replace remove])
  (:require (clojure [string :as string]
                     [set :as set])
            [clojure.java.shell :as sh])
(:use (clojure zip xml))
(:import java.util.Date
         java.text.SimpleDateFormat
         (java.util.concurrent Executors
                               LinkedBlockingQueue)))
```

Вся семантика функций `require`, `refer` и других, остается прежней, но, так как `ns` – это макрос, (обратите внимание, что здесь используются ключевые слова, например, `:use` вместо `use`), отпадает необходимость маскировать (quoting) имена.

> **Примечание.** В предыдущих примерах мы исключили некоторые переменные из пространства имен `clojure.core`, потому что их имена (`next`, `replace` и `remove`) конфликтуют с именами переменных в пространстве имен `clojure.zip`, которое загружается с помощью `use` без наложения дополнительных ограничений несколькими строками ниже. В процессе загрузки `clojure.zip` эти переменные из `clojure.core` были бы переопределены и без нашего участия (с выводом предупреждения), но явное их исключение покажет тем, кто будет сопровождать код в дальнейшем, что мы знали о конфликте.

После объявления, пространства имен можно исследовать и изменять во время выполнения, обычно с помощью REPL. О различных инструментах для работы с пространствами имен во время выполнения рассказывается в разделе «Оболочка REPL», в главе 10.

Пространства имен и файлы

Существует несколько непреложных правил, касающихся организации файлов с исходным кодом на языке Clojure[1]:

---

[1] Эти правила, как и всякие другие, можно игнорировать, если на то есть веские причины, но такие причины встречаются достаточно редко.

Один файл – одно пространство имен. Каждое пространство имен должно быть определено в отдельном файле, а местоположение файла в дереве каталогов с исходным кодом проекта должно соответствовать сегментам в названии пространства имен. Например, код для пространства имен `com.mycompany.foo` должен храниться в файле *com/mycompany/foo.clj*[1]. Когда выполняется загрузка пространства имен с помощью `require` или `use`, например, вызовом `(require 'com.mycompany.foo)`, будет загружен файл *com/mycompany/foo.clj*, после чего пространство имен должно быть определено, в противном случае возникнет ошибка.

Дефисам в названиях пространств имен должны соответствовать подчеркивания в именах файлов. Очень простое правило: если название пространства имен содержит дефисы, как, например, `com.my-project.foo`, исходный код для этого пространства имен должен храниться в файле *com/my_project/foo.clj*. Правило затрагивает только имена файлов и каталогов, соответствующих сегментам в названии пространства имен – в программном коде вы можете продолжать ссылаться на название пространства имен, как оно было объявлено (например, `(require 'com.myproject.foo)`, но не `(require 'com.my_project.foo)`). Это обусловлено тем, что JVM не допускает использование дефисов в именах классов или пакетов, но в Clojure более идиоматично использовать дефисы вместо подчеркиваний в именах программных компонентов, включая пространства имен, переменные, локальные привязки и так далее.

Каждое пространство имен должно начинаться с всеобъемлющей формы ns. Первой формой в «корневом» (и обычно единственном) файле каждого пространства имен должна быть форма `ns`; простые функции управления пространствами имен, такие как `require` и `refer`, обычно используются только в оболочке REPL. Кроме прямого предназначения для применения в программах, использование макроса `ns`:

1. Способствует объединению иначе разрозненных вызовов `require` и других функций.
2. Благодаря размещению в начале файла, упрощает чтение и сопровождение кода, позволяя немедленно получить полное представление о связях и зависимостях пространства имен.

---

[1] Здесь указываются относительные пути, откладываемые от корня дерева каталогов с исходным кодом. Подробнее о физическом размещении файлов проектов на языке Clojure будет рассказываться в разделе «Местоположение, местоположение, местоположение» ниже.

3. Оставляет открытой возможность для рефакторинга и применения других инструментов управления исходным кодом, позволяя изменять списки загружаемых пространств имен, функций и импортируемых классов, потому что `ns` – это макрос, который принимает только имена, не подвергавшиеся дополнительной интерпретации[1]. Применение низкоуровневых форм управления пространствами имен делают применение таких инструментов невозможным.

Избегайте циклических зависимостей в пространствах имен. Зависимости между пространствами имен в любых приложениях на языке Clojure должны образовывать ориентированный нециклический граф. То есть, пространство имен *X* не должно зависеть от пространства имен *Y*, которое само зависит от пространства имен *X* (прямо или косвенно, через промежуточные зависимости). Попытка образовать циклическую зависимость приведет к ошибке:

```
#<Exception java.lang.Exception:
  Cyclic load dependency:
  [ /some/namespace/X ]->/some/namespace/Y->[ /some/namespace/X ]>
```

Используйте `declare` для опережающих ссылок. Clojure последовательно загружает каждую форму из каждого файла пространства имен, разрешая ссылки на ранее объявленные переменные по мере движения вперед. Это означает, что ссылка на необъявленную переменную вызовет ошибку:

```
(defn a [x] (+ constant (b x)))
;= #<CompilerException java.lang.RuntimeException:
;=   Unable to resolve symbol: constant in this context,
;=   compiling:(NO_SOURCE_PATH:1)>
```

Многие языки программирования определяют единицы компиляции, позволяющие отыскивать внутри программы все «потерявшиеся» идентификаторы перед разрешением ссылок на них. Clojure не принадлежит к их числу. Однако не все потеряно, если ради ясности или следования определенному стилю вам потребуется определить более высокоуровневые функции выше используемых ими низко-

[1] Например, инструмент *slamhound* определяет, какие пространства имен необходимо загрузить и какие классы импортировать в форме `ns`, исследуя программный код файле: https://github.com/technomancy/slamhound.

уровневых функций: используйте `declare` для опережающего объявления переменных в текущем пространстве имен, затем определите свои высокоуровневые функции (ссылающиеся на предварительно объявленные переменные), а затем – переменные, объявленные выше:

```
(declare constant b)
;= #'user/b
(defn a [x] (+ constant (b x)))
;= #'user/a
(def constant 42)
;= #'user/constant
(defn b [y] (max y constant))
;= #'user/b
(a 100)
;= 142
```

Единственная проблема, о которой следует помнить, состоит в том, что если вы забудете определить ранее объявленную переменную, при обращении к ней во время выполнения будет возвращаться бессмысленное значение, что практически всегда будет приводить к появлению исключения.

Избегайте давать пространствам имен названия из одного сегмента. Названия пространств имен должны состоять из нескольких сегментов; например, пространство имен `com.my-project.foo` состоит из трех сегментов. В основе этого требования лежат две причины:

1. Если попытаться заранее скомпилировать пространство имен с названием из одного сегмента, в результате будет получен как минимум один файл класса в пакете по умолчанию (то есть, «голый» класс, не включенный в пакет Java). В некоторых окружениях это обстоятельство может препятствовать загрузке пространства имен, и всегда будет препятствовать возможности использования соответствующего класса из Java, из-за ограничений на использование классов в пакете по умолчанию.
2. Даже если вы совершенно уверены, что никогда не будете распространять файлы предварительно скомпилированных классов со своим пространством имен, название которого состоит из одного сегмента, риск конфликтов все равно остается неоправданно высоким, независимо от того, насколько изобретательны вы были при выборе имен.

Не думайте, что мы советуем опускаться до абсурда, разбивая названия пространств имен на сегменты; никому не понравятся име-

на, такие как `com.foo.bar.baz.factory.factory.factories.Factory`. Между подобными и односегментными именами, подверженными риску конфликтов, такими как `app` или `util`, всегда можно найти золотую середину

Независимо от того, как вы организуете свои пространства имен, они (а также весь остальной код и ресурсы, от которых зависит ваша библиотека или приложение) в конечном счете будут загружаться с использованием списка путей *classpath*.

Знакомство с classpath

Для программистов, не знакомых с Java, *classpath* часто является источником заблуждений. classpath – это список путей, который JVM будет использовать для поиска библиотек и ресурсов, определяемых пользователем. В этот список можно включать каталоги, *.zip*-архивы и *.jar*-файлы. Язык Clojure, как опирающийся на JVM, также унаследовал от Java и систему classpath.

Механизм classpath имеет определенные характерные особенности, но они не уникальны и во многом схожи с особенностями других механизмов поиска в списке путей, с которыми вы наверняка знакомы. Например, командные оболочки (shells) в обеих системах, Unix и Windows, определяют переменную окружения `PATH`, которая хранит список путей в файловой системе, где могут храниться выполняемые файлы. Ruby и Python тоже снабжены механизмом поиска в списке путей: в Ruby этот список хранится в переменной времени выполнения `$LOAD_PATH`[1], а Python опирается на переменную окружения `PYTHONPATH`. Во всех этих случаях список путей поиска обрабатывается автоматически, с применением общесистемных настроек и инструментов управления зависимостями (таких как Ruby Gems или `easy_install` и `pip` в Python).

Автоматическая настройка списка путей classpath может быть выполнена с помощью Leiningen и Maven, инструментов, наиболее часто используемых для управления зависимостями в проектах на языке Clojure, а также с помощью популярных сред разработки для Java и Emacs. Например, как только вы определите свои зависимости в файле *project.clj* или *pom.xml*, запуск оболочки REPL с помощью любого из этих инструментов повлечет автоматическое добавление зависимостей в список classpath оболочки REPL. То же происходит при использовании расширений Leiningen или Maven для загрузки

---

[1] Также известной как `$:`.

приложения, например, при запуске веб-приложения на локальном компьютере с помощью `lein-ring` или `jetty:run`[1].

Однако, если вам потребуется запустить Java-процесс непосредственно из командной оболочки (shell), вам придется сконструировать список classpath вручную. Даже если вы не собираетесь запускать программы на языке Clojure из командной строки, знание особенностей определения списка путей classpath поможет вам понять, что именно делают самые совершенные инструменты за вас.

Определение списка путей classpath. По умолчанию список classpath пуст. Он имеет одно неудобное отличие от других механизмов поиска в списке путей, упомянутых выше, которые все по умолчанию включают текущий рабочий каталог (`.`), благодаря чему библиотеки, находящиеся там, будут обнаруживаться во время выполнения.

Определить список путей classpath для Java-процесса можно в командной строке, с помощью флага `-cp`. Например, чтобы в системе Unix включить в список путей поиска текущий рабочий каталог, каталог *src*, файл архива *clojure.jar* и все *.jar*-файлы в каталоге *lib*, можно выполнить следующую команду:

```
java -cp '.:src:clojure.jar:lib/*' clojure.main
```

> **Внимание.** Как и все остальные механизмы поиска в списке путей, classpath определяется системно-зависимым способом, из-за различий соглашений об именовании файлов в разных системах. В Unix-подобных системах компоненты списка путей classpath разделяются двоеточием (`:`); в Windows, используется точка с запятой (`;`). Поэтому пример списка classpath для Unix-подобных систем, представленный выше, в Windows должен быть преобразован в:
>
> ```
> '.;src;clojure.jar;lib*'
> ```

Список classpath и REPL. Список путей classpath доступен программам на языке Clojure во время выполнения:

```
$ java -cp clojure.jar clojure.main
Clojure 1.3.0
(System/getProperty "java.class.path")
;= "clojure.jar"
```

[1] См. раздел «Запуск веб-приложений на локальном компьютере» в главе 17.

Основной список classpath (хранящийся в системном свойстве `java.class.path`) инициализируется при запуске процесса JVM значением параметра командной строки или переменной окружения, но, к сожалению, его нельзя изменить во время выполнения. Это противоречит привычному циклу разработки на языке Clojure, когда окно с оболочкой REPL открывается один раз и остается открытым продолжительное время. Для изменения списка classpath требуется перезапустить JVM и, соответственно, перезапустить REPL[1].

Местоположение, местоположение, местоположение

Существует два основных соглашения о структуре каталогов по умолчанию, используемых в проектах на языке Clojure, которые определяются основными инструментами сборки[2].

Первое – так называемый «стиль Maven», когда все файлы с исходным кодом помещаются в каталог *src*, в разные подкаталоги, в зависимости от языка и роли файлов в проекте. Основной исходный код, определяющий общедоступные API и реализующий функциональные возможности, помещается в каталог *src/main*; код, определяющий модульные и функциональные тесты, который обычно не распространяется, помещается в каталог *src/test*, и так далее:

---

[1] Существует возможность обойти это ограничение. В самом языке Clojure имеется функция `add-classpath`, однако она считается устаревшей и не рекомендуется к использованию. Другой способ заключается в использовании инструмента *pomegranate* (https://github.com/cemerick/pomegranate), являющегося более современной и поддерживаемой заменой функции `add-classpath`, который обеспечивает возможность добавления в окружение времени выполнения *.jar*-файлов и зависимостей, определяемых инструментами Leiningen/Maven. Наконец, все разновидности систем управления модулями для JVM, включая OSGi, NetBeans и серверы приложений всех мастей, предоставляют простой способ дополнения или переопределения списка classpath внутри приложений и отдельных модулей. Причем, все эти механизмы используют средства, встроенные в JVM (такие как управляемые иерархии `ClassLoader`).

[2] Все (основные) инструменты сборки (включая Leiningen и Maven) позволяют размещать файлы с исходным кодом, как вам пожелается. Предлагаемые ими структуры каталогов – всего лишь структуры по умолчанию, хотя трудно представить причины, по которым стоило бы отказаться от этих структур.

Пример 8.1. Структура каталогов проекта «в стиле Maven»

```
<project dir>
 |
 |- src
    |- main
       |- clojure
       |- java
       |- resources
       |- ...
    |- test
      |- clojure
      |- java
      |- resources
      |- ...
```

При такой организации, корень дерева каталогов с исходным кодом на языке Clojure помещается в каталог *src/main/clojure*, с исходным кодом на языке Java[1] – в каталог *src/main/java*, и так далее. Отражение ролей и типов файлов в структуре каталогов может упростить некоторые операции с ними. Например, вместо использования фильтров для выбора файлов определенного типа, можно просто ссылаться на каталог, куда помещаются файлы данного типа. Это существенно упрощает упаковку ресурсов: если имеется набор ресурсов, которые должны включаться в веб-приложение (изображения, сценарии на JavaScript, и так далее), их можно сгруппировать в каталоге *src/main/webapp*, а другие ресурсы, которые не должны распространяться, разместить в другом каталоге, который не будет участвовать в процессе сборки или упаковки.

Структура каталогов в стиле Maven считается наиболее стандартной – проекты, следующие этому стилю, редко отклоняются от принятых соглашений. Основной недостаток организации каталогов в стиле Maven заключается в более длинных путях к файлам из-за использования префиксов *src/main*, *src/test*, и так далее.

Второй основной стиль организации каталогов проектов охарактеризовать несколько сложнее, потому что он может существенно изменяться от проекта к проекту:

---

[1] Здесь предполагается, что исходный код на этом языке имеется; загляните в раздел «Сборка гибридных проектов» ниже, где даются советы для тех, кто разрабатывает гибридные проекты с исходным кодом на языках Java/Clojure.

Пример 8.2. Примеры «свободной» структуры каталогов проекта

```
<project dir>
 |
 |- src
    |- test

<project dir>
 |
 |- src
    |- java
    |- clojure
 |- test
 |- resources
 |- web
```

В отличие от стиля Maven, свободная организация каталогов позволяет уменьшить длину путей к файлам (и тем самым упростить обращение к ним из командной строки) и в целом имеет меньше соглашений, которым следуют почти все проекты, помимо наличия каталогов *src* и *test*. Исходные файлы разных типов часто смешиваются в одном каталоге (например, файлы с исходным кодом на обоих языках, Java и Clojure, могут храниться вместе, в каталоге *src*), хотя иногда это не зависит от выбранного стиля организации проекта. Обычно такая форма организации используется вместе с инструментами сборки, отличными от Maven, включая Leiningen.

Организация программного кода по функциональным признакам

До сих пор мы говорили исключительно о механистических правилах – где должны сохраняться файлы, какие имена следует выбирать, как обозначать соответствие между именами файлов и пространствами имен, и так далее. Более сложными выглядят вопросы, касающиеся организации программного кода на языке Clojure с функциональной точки зрения:

1. Сколько функций должно использоваться для реализации конкретного алгоритма?
2. Сколько функций должно содержаться в пространстве имен?
3. Сколько пространств имен должно содержаться в проекте?

Обычно на эти вопросы проще ответить, если они задаются в связи с другими языками программирования, отчасти потому, что зачастую имеются конкретные требования, явно определяемые «хорошим стилем». Многие фреймворки в других языках определяют, сколько дополнений/компонентов/моделей/расширений должно объявляться (например, «один класс на таблицу в базе данных» или «один модуль на компонент пользовательского интерфейса»), вследствие чего организация программного кода в значительной степени определяется случайными или механистическими характеристиками используемых библиотек и широтой окружения, куда осуществляется развертывание.

В языке Clojure, напротив, организация приложений редко является следствием использования какой-либо библиотеки или фреймворка[1]. В частности, широкое применение приемов функционального программирования и разумное использование макросов позволяют структурировать библиотеки и приложения на языке Clojure так, что они будут отражать контуры предметной области намного отчетливее, чем это возможно в других языках. Справедливости ради следует отметить, что Clojure способствует формированию более ясного представления о предметной области, чем вы имели возможность в течение многих лет, в результате чего структура программ более естественным образом определяется данными и моделями, чем вы могли себе представить.

То есть, можно сказать, что кроме самых общих принципов в Clojure не существует такого понятия, как «типичная структура» программ. Кого-то это может расстроить, кого-то порадовать, в зависимости от опыта и ожиданий. Со своей стороны, мы считаем это обстоятельство живительным, дающим возможность сосредоточиться на основных особенностях, алгоритмах или предметных областях, не отвлекаясь на посторонние детали, которые могут сопровождать тот или иной порядок, давно установленный принятыми решениями, далекими от проблем, которые мы пытаемся решить в своем коде.

---

[1] Даже при расширении или встраивании существующей Java-библиотеки или фреймворка с помощью Clojure, часто сохраняется возможность изолировать операции, связывающие данные и функции на языке Clojure с фреймворком, и организовать программный код наиболее подходящим для выбранной предметной области или архитектуры образом.

Основные принципы организации проектов

С нашей стороны было бы невежливым расплывчато поговорить об общих принципах и не обозначить хотя бы некоторые из них.

- Разные вещи храните отдельно, например, в разных пространствах имен: весь код, занимающийся обработкой пользовательских записей, вероятно, лучше хранить в одном пространстве имен, отдельно от пространства имен, реализующего загрузку шаблонов для отображения веб-содержимого.
- Взаимосвязанные вещи храните вместе, например, сгруппировав их в естественные категории и оформив эти категории как пространства имен. Для этого можно организовать иерархию из названий пространств имен, отражающую отношения между API верхнего уровня (например, `foo.ui`) и нижнего или API провайдера (например, `foo.ui.linux` и `foo.ui.windows`).
- Переменные, содержащие данные, касающиеся особенностей реализации, всегда старайтесь объявлять приватными, добавляя метаданные `^:private` (а приватные функции объявляйте с помощью формы `defn-`). Это убережет клиентов от попадания в зависимость от функциональных особенностей, которые могут изменяться со временем, но сохранит возможность «выхода за кулисы» посредством специальной формы `var` (или синтаксического сахара `#'`) при действительной необходимости.
- Не повторяйтесь: определяйте константы один раз в выделенном для этого пространстве имен и выделите общую функциональность во вспомогательные функции и вспомогательные пространства имен.
- Используйте *абстракции* ссылочных типов, коллекций и последовательностей всегда, когда это возможно. Не полагайтесь на конкретные их реализации.
- Функции с побочными эффектами следует считать вредными и создавать их, только когда это действительно необходимо[1].

---

[1] Чистые функции важны для успешного и эффективного применения приемов функционального программирования, и являются краеугольным камнем в проектировании идиоматических библиотек и приложений на языке Clojure. О функциональном программировании в целом рассказывается в главе 2 и о чистых функциях в частности – в разделе «Чистые функции» в этой же главе.

В целом проекты на языке Clojure только выиграют от модульной организации и разделения задач, как и проекты на любых других языках. Кроме того, помните, что пространства имен – это организационный инструмент, предоставляемый исключительно для вашего удобства: большое приложение, содержащее 500 пространств имен, будет действовать так же и показывать такую же производительность, как и приложение, состоящее из единственного, огромного пространства имен. Поэтому вы можете как угодно структурировать свое приложение, чтобы обеспечить соответствие структуре предметной области и особенностям организации труда в вашей команде.

Сборка

«Сборка» – это обобщающее понятие, с течением времени охватывающее все больше и больше из того, что мы должны сделать *после* того, как код будет написан, но до того, как он будет выпущен (еще один нагруженный термин, учитывая такие сложности, как представление программного обеспечения в виде службы, облачные вычисления и так далее).

В нашем случае под термином *сборка* (build) будут подразумеваться:

- компиляция;
- управление зависимостями, позволяющее использовать внешние библиотеки;
- упаковка результатов компиляции и других ресурсов проекта в *артефакты* (artifacts);
- распространение этих артефактов в контексте управления зависимостями.

Это формальное описание выглядит сложнее, чем действия, которые оно описывает. Вы наверняка уже делали все эти действия, перечисленные в табл. 8.1.

Таблица 8.1. Сравнение решений «сборки» в разных языках программирования

	Компиляция	Управление зависимостями	Упаковка	Распространение
Ruby	rake	`gem`, `rvm`	Gems	rubygems.org
Python	`distutils`, `SCons`	`pip`, `virtualenv`	Eggs	PyPI[a]
Java	`javac`, Ant, Maven, Gradle, и т.д.	Модель Maven, Ivy	jar-файлы и их варианты	Репозитории артефактов Maven

[a] http://pypi.python.org/pypi

Так как Clojure является языком JVM, он естественным образом использует обширный набор инструментов сборки, упаковки и инфраструктуру распространения этой экосистемы:

- Leiningen использует значительную часть инфраструктуры Maven, одновременно обеспечивая более удобный «пользовательский интерфейс» и позволяя использовать навыки разработки на языке Clojure;
- для Maven, Gradle и Ant существуют расширения, позволяющие управлять сборкой программ на языке Clojure из этих инструментов;
- библиотеки на языке Clojure упаковываются в *.jar*-файлы, веб-приложения упаковываются (обычно) в *.war*-файлы[1], и так далее;
- библиотеки на языке Clojure распространяются через репозитории Maven, доступные всем инструментам сборки для Java (и, соответственно, для Clojure).

Такое совпадение инструментов и приемов сборки между Clojure и Java позволяет использовать Java-библиотеки в приложениях на языке Clojure распространять библиотеки на языке Clojure, которые могут использоваться в программах на других языках JVM (таких как Java, Groovy, Scala, JRuby, Jython и так далее).

Если вам уже приходилось пользоваться Java или какими-то другими языками JVM, вы увидите, что добавление программного кода на Clojure в свои проекты оказывает минимальное влияние на привычный процесс сборки. С другой стороны, если прежде вы использовали Ruby, Python или какой-то другой язык, не имеющий отношения к JVM, вас должно утешить то обстоятельство, что процесс сборки и настройки программного кода на языке Clojure практически всегда проще, чем на Java.

Предварительная компиляция

Как упоминалось в разделе «Интерактивная оболочка REPL для Clojure», в главе 1, исходный программный код на языке Clojure *всегда* компилируется, потому что не существует интерпретатора Clojure. Компиляция, в результате которой генерируется байт-код для компилируемого исходного кода, и загрузка этого байт-кода в виртуальную машину JVM могут выполняться двумя разными способами.

---

[1] Инструмент Heroku стоит особняком в этом отношении, см. раздел «Clojure на платформе Heroku» в главе 20.

- Во время выполнения. Именно так и происходит, когда вы пользуетесь оболочкой REPL или загружаете файл с исходным программным кодом с диска. Содержимое файлов компилируется в байт-код, который далее загружается в JVM. Этот байт-код и классы, которые он определяет, не сохраняются после завершения JVM.
- Предварительная (Ahead-of-Time, AOT) компиляция выполняется точно так же, как и компиляция во время выполнения, но полученный байт-код сохраняется на диск в виде файлов классов JVM[1]. Эти файлы классов затем могут повторно использоваться другими процессами JVM вместо оригинальных файлов с исходным кодом на языке Clojure.

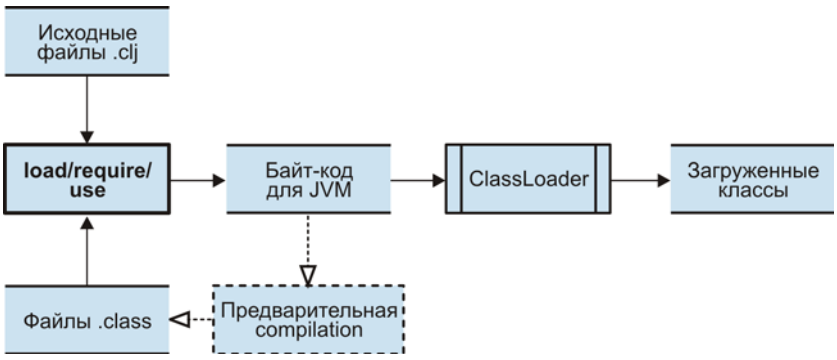

Рис. 8.1. Процесс компиляции в языке Clojure

Код, загруженный из файлов с исходным кодом и из файлов классов, созданных в результате предварительной компиляции, не имеет функциональных отличий. Операция загрузки пространства имен – например, `(require 'clojure.set)` – отыщет в списке путей classpath файл *clojure/set.clj* с исходным кодом, определяющим это пространство имен, или соответствующие файлы классов, скомпилированные предварительно[2].

---

[1] Это в точности соответствует тому, как работает компилятор javac, который сохраняет файлы классов на диск, сгенерированные на основе содержимого файлов с исходным программным кодом на языке Java.

[2] Если доступными окажутся и исходный файл, и файлы классов для указанного пространства имен, предпочтение будет отдано файлам классов, если они окажутся более новыми, чем файл с исходным кодом. Это дает

Таким образом, за исключением отдельных случаев, предварительная компиляция в целом является необязательной. Так как библиотеки и приложения на языке Clojure обычно не требуют предварительной компиляции, разумнее распространять исходные файлы на языке Clojure, если это возможно. В действительности, распространение предварительно скомпилированных файлов имеет ряд существенных недостатков:

1. Файлы классов для JVM по определению занимают больше места на диске, чем исходные файлы на языке Clojure, из которых они генерируются.
2. Предварительная компиляция добавляет еще один шаг в цикл разработки.
3. Предварительная компиляция тесно связывает библиотеку или приложение на языке Clojure с версией Clojure, использованной для предварительной компиляции. Нельзя уповать, что код, скомпилированный в одной версии Clojure, будет развернут во время выполнения другой версией Clojure.
4. Предварительная компиляция является *транзитивной* операцией. Если вы скомпилировали пространство имен `foo`, которое загружает пространство имен `bar`, тогда пространство имен `bar` так же будет скомпилировано, и так далее. В зависимости от особенностей конкретного пространства имен внутри вашей библиотеки или приложения, это может привести к компиляции большего числа файлов, чем вы предполагали.

Примечание. Если вам требуется всего лишь скомпилировать один или несколько классов, служащих точкой входа для Java, можно воспользоваться простым способом разрыва транзитивности: используйте форму `gen-class` (или параметр `:gen-class` макроса `ns`) с единственной реализацией пространства имен, указанной в параметре `:impl-ns`. В процессе компиляции такие зависимости пропускаются, поэтому компиляция вызова `gen-class` для загрузки пространства имен не повлечет компиляцию всего программного кода.

Предварительную компиляцию следует использовать лишь для отдельных случаев:

возможность включать в список classpath деревья каталогов с исходными текстами и с предварительно скомпилированными файлами, и по мере необходимости перезагружать пространства имен из изменившихся исходных файлов в ходе сеанса работы с оболочкой REPL.

1. Когда вы не можете или не желаете распространять исходный код.
2. Когда вы желаете использовать различные инструменты для обработки файлов классов в процессе упаковки, например, с целью обфускации.
3. Когда время загрузки приложения играет *чрезвычайно важную роль* и вы готовы платить за это неудобствами, сопряженными с предварительной компиляцией. Загрузка скомпилированных файлов классов выполняется намного быстрее, чем загрузка и компиляция файлов с исходным кодом.
4. Когда предполагается, что программный код на языке Clojure будет использоваться из программ на Java или другом языке JVM посредством Java-классов или интерфейсов, сгенерированных формой `gen-class`, `defrecord`, `deftype` или `defprotocol`[1].

О некоторых дополнительных возможностях инструментов Leiningen и Maven, связанных предварительной компиляцией, будет рассказываться в разделе «Настройка предварительной компиляции» ниже.

Управление зависимостями

В любой команде, занимающейся разработкой программного обеспечения, приходится использовать инструменты управления зависимостями в своих проектах. Команды, использующие Clojure, ничем не отличаются в этом смысле.

Еще не забыты темные времена, когда считалось нормой передавать зависимости в виде файлов *.zip* и *.tar.gz*, вручную распаковывать их в нужные каталоги проекта (обычно в каталог *lib*) и отправлять результат в систему управления версиями. После этого инструмент сборки мог обращаться к этим зависимостям, используя простые пути к файлам, и либо связывать эти зависимости в двоичные дистрибутивы или другие артефакты, либо беспечно предполагать, что пользователи проекта уже будут иметь те же версии тех же зависимостей или знать, как получить их.

К счастью современные требования отвергают такой подход, основанный на настройке вручную и полной неповторимости процесса.

В языке Perl имеется CPAN (Comprehensive Perl Archive Network – всеобъемлющий сетевой архив ресурсов для Perl, являющий-

---

[1] В общем случае имеет смысл ограничиться предварительной компиляцией только тех пространств имен, которые включают перечисленные формы, чтобы уменьшить размер упаковываемых артефактов.

ся, пожалуй, дедушкой всех систем управления зависимостями), в Python – `pip` и `virtualenv`, в Ruby – `gem` и `rvm`, а в JVM лидирующие позиции в управлении зависимостями занимает Maven и модель, разработанная в середине 2000-х. Clojure полностью поддерживает эту модель, независимо от инструментов сборки, которые вы будете использовать.

Модель Maven управления зависимостями

Модель управления зависимостями, реализованная в инструменте Maven, обеспечивает:

- идентификация и управление версиями *артефактов* с использованием *координат* (coordinates);
- объявление *зависимостей* артефактов и проектов, создающих их;
- сохранение артефактов в *репозиториях* и извлечение их оттуда вместе с описаниями зависимостей;
- вычисление *транзитивных зависимостей* артефактов;

Рассмотрим поближе эти понятия и коротко опишем механизмы, вовлекаемые ими.

Артефакты и координаты

Артефакт – это любой файл, являющийся продуктом процесса сборки проекта. Библиотеки и приложения на языке Clojure упаковываются точно так же, как написанные на Java и других языках JVM. Более конкретно это означает, что вы будет использовать и производить артефакты двух основных типов:

- *.jar*-файлы, являющиеся *.zip*-файлами, которые содержат[1]:
  - исходные файлы на языке Clojure, файлы классов JVM и другие ресурсы (такие как статические конфигурационные файлы, изображения и прочие) с сохранением иерархии размещения их в дереве каталогов проекта с исходными файлами или в дереве каталогов цели компиляции;
  - необязательные метаданные в каталоге **`META-INF`**.
- *.war*-файлы, так же являющиеся *.zip*-файлами, – стандартный для JVM способ упаковки веб-приложений; подробнее о *.war*-файлах и их использовании рассказывается в разделе «Упаковка веб-приложений», в главе 17.

---

[1] Более подробную информацию о *.jar*-файлах можно получить по адресу: http://java.sun.com/developer/Books/javaprogramming/JAR/basics.

Редко используемые способы упаковки

В репозиториях Maven также можно встретить самые обычные файлы *.zip* и *.tar.gz*, но они, как правило, используются для распространения ресурсов, не содержащих программный код. Например, вам может потребоваться включить инсталлятор JVM вместе с инсталлятором клиентского приложения, который собирается автоматически на этапе сборки; упаковав инсталлятор JVM и поместив его в репозиторий Maven своей организации, можно было бы гарантировать возможность объявления в вашем проекте зависимости от него и управлять включением этого инсталлятора.

Кроме того, прикладные фреймворки, такие как Eclipse и NetBeans, поддерживают свои способы упаковки и предъявляют свои требования, которые, впрочем, являются всего лишь вариациями на тему *.jar*-файлов и редко применяются (если вообще применяются) за пределами фреймворков, для которых они предназначены.

Идентификация артефактов всех типов выполняется с помощью *координат* (coordinates) , комплекса признаков, совокупность которых уникально идентифицирует определенную версию артефакта:

- `groupId` – обычно идентификатор организации или проекта, такой как like `org.apache.lucene` или `com.google.collections`;
- `artifactId` – идентификатор артефакта внутри организации или проекта, такой как `lucene-core` или `lucene-queryparser`; проекты часто производят множество связанных между собой артефактов с одним и тем же идентификатором `groupId`, хотя для небольших открытых библиотек на языке Clojure обычно принято использовать одинаковые `groupId` и `artifactId`, если между организацией и проектом нет никаких различий;
- `packaging` – идентификатор типа артефакта, являющийся ссылкой, которая соответствует расширению файла самого артефакта; По умолчанию имеет значение `jar` и, как правило, не определяется для случаев по умолчанию;
- `version` – строка с номером версии, которая в идеале должна следовать соглашениям о семантическом управлении версиями[1].

---

[1] Семантическое управление версиями – это широко распространенное соглашение о порядке определения величины изменений между версиями программного продукта. Подробнее об этом можно прочитать по адресу: http://semver.org (аналогичную статью на русском языке можно найти по адресу: http://habrahabr.ru/post/134033/#habracut. – *Прим. перев.*).

В текстовом представлении координаты Maven часто определяются в формате: `groupId:artifactId:packaging:version`, то есть, версия v1.3.0 артефакта Clojure в виде *.jar*-файла будет идентифицироваться, как `org.clojure:clojure:1.3.0` (напомню, что признак `packaging` по умолчанию получает значение `jar`). Каждый проект определяет собственные координаты – иногда в файле *pom.xml*, при использовании Maven, иногда в файле *project.clj*, при использовании Leiningen. В любом случае, каждый раз, когда выполняется сборка проекта и автор желает распространить полученные артефакты, вместе с артефактами в репозиторий Maven выгружается и соответствующий файл *pom.xml*.

Репозитории

Когда комплект артефактов выгружается в репозиторий Maven, этот репозиторий обычно индексирует информацию о версии и о зависимостях, находящуюся в файле *pom.xml*, выгруженном вместе с артефактами; это называют *развертыванием* (deployment). С этого момента репозиторий сможет передавать артефакты любым клиентам, желающим получить их (практически всегда – другим разработчикам, чьи проекты зависят от данных артефактов).

Во всем мире существуют сотни (если не тысячи) общедоступных репозиториев, но крупных, хранящих большинство артефактов Maven, совсем немного.

- *Maven central* – самый крупный репозиторий, используемый по умолчанию инструментами сборки на основе Maven для поиска зависимостей. Все официальные дистрибутивы Clojure и основные библиотеки хранятся в этом репозитории.
- *Clojars.org* – репозиторий открытого сообщества Clojure, для которого в Leiningen предусмотрена упрощенная интеграция. В репозитории Clojars хранятся многие популярные открытые библиотеки для Clojure, включая Ring, Clutch и Enlive.
- Крупные организации, развивающие проекты с открытыми исходными кодами, такие как Apache, JBoss и RedHat, поддерживают собственные репозитории Maven.

В некоторый момент вам, возможно, доведется работать с двумя другими типами репозиториев Maven.

- Некоторые компании и организации поддерживают собственные частные/внутренние репозитории для хранения артефактов, создаваемыми их собственными проектами, и иногда проксируют общедоступные репозитории, такие как Maven central и Clojars.

- Локальный репозиторий, создаваемый инструментом Maven и другими инструментами сборки на его основе в каталоге *~/.m2/repository*. Именно сюда загружаются и сохраняются зависимости для ваших проектов. При желании этот репозиторий можно выбрать для *установки* артефактов, производимых в процессе сборки проектов (концептуально это тот же самый процесс, что и развертывание (deployment), но получивший такое название, чтобы отличать установку в локальный репозиторий).

Зависимости

Каждый проект определяет не только свои координаты, но и свои зависимости. *Зависимости* выражаются как ссылки на артефакты из других проектов с использованием координат этих артефактов. Опираясь на описания зависимостей, Maven и другие инструменты сборки на основе Maven могут.

- Определять множество транзитивных зависимостей проекта, то есть, зависимостей от зависимостей проекта, и так далее. Говоря более формальным языком, зависимости проекта имеют форму ориентированного нециклического графа с корнем в проекте, который собирается, запускается, тестируется и так далее. Образование циклических зависимостей считается фатальной ошибкой.
- На основе полного множества транзитивных зависимостей проекта запускать компиляцию или новые процессы REPL и приложения, с зависимостями, добавленными в список classpath JVM, что позволяет этим процессам ссылаться на классы, ресурсы и исходные файлы на языке Clojure, содержащиеся в зависимостях.

Разрешение зависимостей проекта сводится к обходу графа, о чем вам знать совсем необязательно[1]. Единственное, о чем следует помнить – гибкость определения версий зависимостей, выражающаяся, прежде всего, в возможности указывать *версии срезов*, но и *диапазоны версий*.

---

[1] Такие инструменты, как Leiningen и Maven, автоматически решают все проблемы. Если вы хотите узнать больше о разрешении зависимостей в модели Maven, обращайтесь к руководству: http://docs.codehaus.org/display/MAVEN/Dependency+Mediation+and+Conflict+Resolution.

Clojure – «всего лишь» одна из зависимостей

Читатели с опытом использования Python, Ruby и других языков, имеющих собственную среду времени выполнения, часто полагают, что Clojure «устанавливается» так же, как, например, какая-то определенная версия Ruby. Однако это не так.

С технической точки зрения, Clojure – это всего лишь библиотека Java/JVM и, соответственно, всего лишь одна из зависимостей вашего проекта. Устанавливать необходимо JVM. После этого можно будет добавить Clojure в список classpath приложения, что обычно делается с помощью Leiningen или Maven во время разработки и тестирования.

Тот факт, что Clojure является библиотекой для JVM, может иметь весьма ощутимые выгоды, в терминах развертывания[1] и получения[2].

Версии срезов и выпусков. В модели Maven управления зависимостями, версии могут быть классифицированы или как *версии срезов* (snapshot version), или как версии выпусков (release version). В большинстве случаев номером версии является строка. Например, строки: `1.0.0`, `3.6`, `0.2.1-beta5`, и так далее, считаются версиями выпусков, при этом предполагается, что эти версии фиксированы во времени и никогда не будут изменяться. Это гарантируется всеми нормальными репозиториями, которые не позволят изменить артефакт, для которого указана строка с номером версии выпуска. Эта особенность обеспечивает повторяемость сборки: так как артефакты не изменяются после их развертывания в репозитории, то после успешной сборки и проверки приложения с определенной версией зависимости можно быть уверенными, что вы всегда сможете получить те же результаты.

Версии срезов совершенно иные. Примечательной особенностью строк с номерами версий срезов является окончание `-SNAPSHOT`, например: `1.0.0-SNAPSHOT`, `3.6-SNAPSHOT` или `0.2.1-beta5-SNAPSHOT`. Версии срезов предназначены для идентификации артефактов, находящих-

---

[1] Когда дело доходит до развертывания приложения, на выбор имеется несколько вариантов упаковки, включая: *.war*-файлы для веб-приложений, обсуждаемые в разделе «Упаковка веб-приложений», в главе 17; разновидности *.jar*-файлов для развертывания обычных приложений и непосредственно поддерживаемые инструментом Leiningen, описываемом в разделе «Leiningen» ниже; и другие методы, каждый из которых стремится создать единственный сборный файл, содержащий весь код проекта и его транзитивных зависимостей, включая Clojur.

[2] См. раздел «Clojure – всего лишь .jar-файл».

ся в процессе разработки. То есть, один и тот же номер версии может соответствовать разным вариантам реализации артефакта в разные моменты времени, по мере того, как будут создаваться и сохраняться в репозитории версии для разработчиков.

Например, допустим, что вы желаете следить за процессом разработки, результатом которого будет версия `2.0.0` некоторой библиотеки, потому что в этой версии будут реализованы некоторые важные особенности, необходимые вашему проекту. В репозитории наверняка будет сохраняться множество сборок этой библиотеки с версией `2.0.0-SNAPSHOT`, и если вы укажете эту строку версии в зависимостях вашего проекта, вы всегда будете получать и использовать самую последнюю разрабатываемую версию[1]. Как только авторы завершат работу, вы сможете перейти на использование версии выпуска `2.0.0` библиотеки и развернуть окончательную версию своего артефакта в репозитории проекта.

Диапазоны версий. Допустим, у вас имеется зависимость от версии 1.6.0 некоторой библиотеки, но вы знаете, что можно смело использовать следующие версии этой библиотеки, вплоть до смены старшего номера версии, так как в этих версиях сохраняется совместимость API. Поскольку почти для всех артефактов используется семантическое управление версиями, как в данном случае, часто бывает полезно определить зависимость не от конкретной версии, а от *диапазона версий*. В табл. 8.2 перечислены форматы определения диапазонов версий, поддерживаемые инструментом Maven.

Таблица 8.2. Форматы диапазонов версий, поддерживаемые Maven[2]

Формат диапазона	Семантика
(,1.0]	x <= 1.0
1.0	«Мягкое» требование версии 1.0
[1.0]	Жесткое требование версии 1.0
[1.2,1.3]	1.2 <= x <= 1.3
[1.0,2.0)	1.0 <= x < 2.0
[1.5,)	x >= 1.5

---

[1] Обычно Maven и другие инструменты на его основе проверяют появление новых срезов каждые 24 часа. Если вы пользуетесь инструментом Maven, передайте ему ключ `-U`, чтобы принудительно выполнить проверку обновлений.

[2] Взято из http://docs.codehaus.org/display/MAVEN/Dependency+Mediation+and+Conflict+Resolution.

То есть мы можем объявить зависимость от версии библиотеки, как `[1.6.0,2.0.0)`, и позволить артефактам нашего проекта использовать библиотеку любой версии, в диапазоне от `1.6.0` (включительно) до `2.0.0` (исключительно). Когда будет выпущена версия `2.0.0` библиотеки (где, возможно, будет нарушена обратная совместимость с версиями `1.x.x`), нам потребуется проверить (и, может быть, поправить) работоспособность проекта с версией `2.0.0` и выпустить новую его версию, возможно, определив зависимость от диапазона версий библиотеки, как `[2.0.0, 3.0.0)`.

Внимание. Обратите внимание, что в табл. 8.2 «голый» номер версии (например, `1.0`) означает «мягкое» требование соответствия номера версии. Это означает, что если в зависимостях артефакта присутствуют два «голых» номера версий – например, сам артефакт зависит от версии `1.2` библиотеки, а одна из его зависимостей в свою очередь зависит от версии `1.6` той же библиотеки – во время сборки, в оболочке REPL и так далее, будет выбрана и использована более новая версия. Это почти никогда не вызывает проблем, но если они возникнут, укажите диапазон версий, чтобы устранить неоднозначность. В данном случае, если явно указать диапазон версий библиотеки, как `[1.2,1.5]`, будет выбрана версия `1.5`.

Возможность такого сценария и применения диапазонов версий в целом зависит от особенностей нумерации версий в проектах, производящих библиотеки, от которых вы зависите. Если в них используется семантическая нумерация версий (того или иного вида), то применение диапазонов версий позволит гарантировать, что артефакты, производимые вашим проектом, будут использоваться только с совместимыми версиями зависимостей. С другой стороны, если нумерация версий зависимостей следует какой-то другой схеме[1], тогда диапазоны версий могут оказаться бесполезной игрушкой.

Инструменты сборки и шаблоны настройки

Теперь, когда мы познакомились с некоторыми ключевыми особенностями организации и сборки проектов, можно рассмотреть не-

[1] Примером такого неудачного подхода может служить выбор дат в качестве номеров версий, таких как `20110705`. Еще хуже, когда в качестве номеров версий используются контрольные суммы, как например SHA в системах контроля версий Git или Mercurial, имеющие вид `8c7be13792`, которые не только не поддерживают понятие диапазонов, но и не гарантируют монотонное увеличение – то есть исходя из координат артефакта нельзя сказать, какая из двух версий новее.

сколько примеров работы с двумя инструментами сборки, наиболее популярными в сообществе Clojure – Leiningen и Maven. Однако, этот раздел следует рассматривать лишь как краткий обзор, поэтому имейте в виду, что:

> **Внимание.** Каждый из этих инструментов имеет множество глубинных особенностей, и здесь не будет даваться исчерпывающее описание всех возможностей и недостатков этих инструментов. Какой бы инструмент сборки вы ни выбрали, обязательно ознакомьтесь с документацией к нему и ресурсами сообщества, чтобы максимально использовать предлагаемые им возможности. Наконец, в главе 17 будут представлены дополнительные примеры сборки веб-приложений.

Не влезайте в то, что работает. Как уже говорилось в начале этой главы, мы не можем претендовать на истину в последней инстанции – особенно, когда дело доходит до выбора инструмента сборки. Тем не менее, существуют некоторые правила, которые помогут вам принять решение о том, какой инструмент выбрать для своих проектов на языке Clojure. Эти правила будут обсуждаться ниже, в разделах с описанием Leiningen и Maven. Однако есть одно правило, которое отменяет все другие правила:

Если в вашей организации считается стандартом использовать какой-то определенный набор инструментов на основе JVM, используйте его. Расширения поддержки языка Clojure существуют для Ant[1], Maven, Gradle[2], Buildr и, возможно, других, не знакомых нам инструментов, поэтому внедрить Clojure в проекты, уже использующие эти инструменты, будет совсем несложно и не потребует вмешиваться в устоявшиеся и хорошо зарекомендовавшие себя процессы сборки проектов и управления ими.

Maven

Maven является, пожалуй, наиболее распространенным инструментом сборки в мире Java. Это открытый проект, развиваемый под эгидой организации Apache Software Foundation[3], область применения которого намного шире, чем большинства других инструментов сборки. Через огромное количество сторонних расширений, он стре-

---

[1] https://github.com/jmcconnell/clojure-ant-tasks.

[2] https://bitbucket.org/kotarak/clojuresque/wiki/Home.

[3] Мы рекомендуем начинать знакомство с инструментом с версии Maven 3, доступной по адресу: http://maven.apache.org.

мится обеспечить управление «всем жизненным циклом» программных проектов, включая интеграционное и функциональное тестирование, оценка охвата кода тестами, создание отчетов по результатам тестирования и управление версиями, в дополнение к типичным задачам сборки, таким как компиляция и упаковка кода.

Одной из отличительных характеристик Maven является превосходная интеграция с другими инструментальными средствами на основе JVM, например, с интегрированными средами разработки, такими как Eclipse, NetBeans и IntelliJ; средствами поддержки коллективной разработки, такими как Hudson/Jenkins, Clover и Cobertura; и вспомогательными инструментами сборки, такими как NSIS, IzPack, `javacc`, ANTLR и Selenium. Если вам нужна максимальная интеграция между этими инструментами и инструментом сборки, или возможность управления «полным жизненным циклом» имеет большое значение для вас или вашей организации, вам стоит задуматься об использовании Maven в своих проектах на языке Clojure.

Ниже демонстрируется содержимое простого файла *pom.xml*, который можно использовать в качестве заготовки описания проектов Clojure в Maven[1]:

Пример 8.3. Простейший файл pom.xml для настройки несложных проектов на языке Clojure

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                         http://maven.apache.org/maven-v4_0_0.xsd">

     <modelVersion>4.0.0</modelVersion>
     <groupId>com.clojurebook</groupId>
     <artifactId>sample-maven-project</artifactId>
     <version>1.0.0</version>
     <packaging>clojure</packaging>

     <dependencies>
```

[1] В Интернете существует огромный объем документации для Maven, включая две бесплатные книги, написанные авторами этого инструмента и доступные по адресу: http://www.sonatype.com/books.html: «Maven by Example» – вводное руководство по Maven, и «Maven: the Complete Reference» – достойная стать настольной книгой для тех, кто желает разобраться в мельчайших деталях инструмента.

```
        <dependency>
            <groupId>org.clojure</groupId>
            <artifactId>clojure</artifactId>
            <version>1.3.0</version>
        </dependency>
    </dependencies>

    <build>
        <resources>
            <resource>
                <directory>src/main/clojure</directory>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <groupId>com.theoryinpractise</groupId>
                <artifactId>clojure-maven-plugin</artifactId>
                <version>1.3.8</version>
                <extensions>true</extensions>
                <configuration>
                    <warnOnReflection>true</warnOnReflection>
                    <temporaryOutputDirectory>true</temporaryOutputDirectory>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

В этом файле *pom.xml*:

- определяются координаты Maven `com.clojurebook:sample-maven-project:1.0.0`;
- определяется способ упаковки `clojure`, который требует от расширения `clojure-maven-plugin` добавить цели для предварительной компиляции и модульного тестирования в соответствующие фазы Maven;
- определяется единственная зависимость от Clojure v1.3.0 (`org.clojure:clojure:1.3.0`);
- добавляется стандартный каталог с исходными текстами на языке Clojure в виде каталога ресурсов Maven; это обеспечит добавление исходных файлов из этого каталога в *.jar*-файл, создаваемый в результате сборки проекта;
- настраивает расширение `clojure-maven-plugin` на запуск предварительной компиляции в режиме «проверки целостности»

(sanity check), как описывается в разделе «Настройка предварительной компиляции» ниже, в ходе которой выводятся предупреждения при обнаружении обращений к механизму рефлексии и во временном каталоге сохраняются файлы классов (чтобы исключить их добавление в упакованные артефакты).

Ниже приводятся некоторые типичные операции с проектом, вовлекающие этот файл *pom.xml*:

- `mvn clojure:repl` – запустит новый сеанс работы с оболочкой Clojure REPL, которой будет передан список classpath, включающий все транзитивные зависимости проекта;
- `mvn package` – выполнит предварительную компиляцию исходных файлов в режиме проверки целостности и соберет *.jar*-файл, содержащий эти файлы;
- `mvn test` – выполнит все тесты для кода на Java и Clojure, находящиеся в вашем проекте (в каталогах с корнем в *test/src/java* и *test/src/clojure*, соответственно);
- `mvn install` – в дополнение к действиям, выполняемым командой `mvn package`, установит *.jar*-файл в локальный репозиторий Maven;
- `mvn deploy` – в дополнение к действиям, выполняемым командой `mvn install`, развернет *.jar*-файл в удаленном репозитории, который вы можете добавить в файл *pom.xml*.

Вся поддержка языка Clojure в Maven обеспечивается расширением `clojure-maven-plugin`[1], поддерживающим массу параметров для управления предварительной компиляцией, выполнением определенных сценариев в проекте, модульным и функциональным тестированием, и так далее. Кроме того, он поддерживает множество других целей, помимо `clojure:repl`, которые могут пригодиться в процессе разработки.

Кроме `clojure-maven-plugin`, существуют сотни других расширений для Maven, автоматизирующих различные операции, связанные со сборкой, тестированием и управлением проектом, а также обеспечивающих расширенную интеграцию проектов Maven с внешними инструментами и окружениями. Просто, выполните поиск в Интернете по строке `foo maven plugin` и вы найдете множество полезных расширений поддержки `foo`.

---

[1] http://github.com/talios/clojure-maven-plugin.

Leiningen

Leiningen[1] позиционируется как «Инструмент сборки для Clojure, предназначенный уберечь вас от паники»[2]. Этот инструмент появился в основном из-за излишней сложности использования Maven и Ant при решении типичных задач сборки проектов на языке Clojure. Его цель – сделать управление проектом более простым, чем это возможно при использовании Ant или Maven.

Если вы пришли из мира, не имеющего отношения к JVM – из Ruby, Python или других похожих языков – вы наверняка найдете Leiningen более удобным, чем Maven. Даже при том, что за кулисами он использует некоторые части Maven, инструмент Leiningen предоставляет более идиоматичное для Clojure представление модели зависимостей Maven, и более простую процедуру разработки. В частности, изменение или расширение процедуры сборки в Leiningen можно полностью выполнить на языке Clojure[3], что кажется более привлекательным по сравнению с настройкой сборки в Maven. Однако за это приходится платить отсутствием большого количества расширений для Leiningen и возможностей интеграции, предлагаемых экосистемой Maven, хотя количество расширений для Leiningen продолжает увеличиваться[4].

Ниже демонстрируется содержимое простого файла *project.clj*, который можно использовать в качестве заготовки описания проектов Clojure в Leiningen:

Пример 8.4. Простейший файл project.clj для настройки несложных проектов на языке Clojure

```
(defproject com.clojurebook/sample-lein-project "1.0.0"
  :dependencies [[org.clojure/clojure "1.3.0"]])
```

---

[1] Название инструмента Leiningen, который также часто называют как «lein», обычно произносят как «LINE-ing-en» (лайн-инг-ен), хотя нередко можно услышать произношение «LINE-in-gen» (лайн-ин-ген).

[2] Мы рекомендуем начинать знакомство с инструментом Leiningen с версии, не ниже v1.7.0: http://leiningen.org. Все примеры использования Leiningen в этой книге были проверены с версией v1.7.0, а также с недавно вышедшей предварительной версией v2.0.

[3] Пример добавления небольших изменений в процедуру сборки будет представлен в разделе «Сборка гибридных проектов» ниже.

[4] Неполный список расширений для Leiningen можно найти по адресу: https://github.com/technomancy/leiningen/wiki/plugins.

`defproject` – это макрос, определяющий модель проекта для Leiningen. Кроме пар ключ/значение, составляющих большую часть конфигурации проекта, макрос `defproject` требует передачи в первых двух аргументах координат проекта (символа Clojure, и строки – в этом примере `com.clojurebook/sample-lein-project` и `"1.0.0"` – соответствующих координатам `com.clojurebook:sample-lein-project:1.0.0` в нотации Maven).

Макрос `defproject` обеспечивает запись указанного символа в оба параметра, `groupId` и `artifactId`. Это типично для открытых проектов, где название проекта одновременно *является* названием организации; например, в настройках проекта Ring (веб-фреймворк на языке Clojure) в первых двух аргументах макросу `defproject` передаются `ring` и `"1.0.1"`, что в результате дает координаты Maven: `ring:ring:1.0.1`.

> **Примечание.** Leiningen предоставляет команду для создания заготовки нового проекта; вызов `lein new my-project` создаст новый каталог my-project, содержащий файл project.clj (с координатами `my-project "1.0.0-SNAPSHOT"`), и заготовки для файлов с исходным кодом и тестами.

Этот простейший файл *project.clj* определяет практически те же параметры конфигурации, как и пример файла `pom.xml` для Maven, представленный в разделе «Maven» выше, с той лишь разницей, что корень дерева каталогов с исходным кодом на языке Clojure будет находиться в каталоге `src`, а не *src/main/clojure*, а корень дерева с исходным кодом тестов – в каталоге *test*, а не *src/test/clojure*. Настоящие различия между двумя подходами к настройке такого простого проекта состоит в том, что поведение инструмента сборки Leiningen зависит от того, какие команды `lein` и в каком порядке выполняются, тогда как Maven обеспечивает определенный порядок выполнения и семантику фаз сборки.

Кроме этих различий, процедура сборки проекта с помощью Leiningen очень похожа на процедуру сборки с помощью Maven:

- `lein repl` – запустит новый сеанс работы с оболочкой Clojure REPL, которой будет передан список classpath, включающий все транзитивные зависимости проекта;
- `lein test` – выполнит все тесты, находящиеся в вашем проекте (обычно находящиеся в каталогах с корнем в *test*);
- `lein jar` – выполнит такую же операцию упаковки, что и команда `mvn package`, но не будет автоматически выполнять предварительную компиляцию файлов с исходным кодом;

- `lein uberjar` – произведет разновидность *.jar*-файла (uberjar), такой же, который производит команда `lein jar`, но со всеми транзитивными зависимостями проекта, «распакованными» в него; такие файлы обычно используются, чтобы упростить развертывание, так как позволяет распространять все приложение целиком в виде единственного файла, запускаемого единственным вызовом команды `java`[1];
- `lein compile` – выполнит предварительную компиляцию всех файлов с исходным кодом Clojure в проекте, опираясь на параметр настройки `:aot` в *project.clj*; об особенностях настройки предварительной компиляции в Leiningen рассказывается ниже, в разделе «Настройка предварительной компиляции»;
- `lein pom` – сгенерирует Maven-совместимый файл *pom.xml*, содержащий информацию о проекте и его зависимостях, указанную в файле *project.clj*; этот файл *pom.xml* можно использовать для развертывания в удаленном или для установки в локальном репозитории Maven;
- `lein deps` – обеспечит доступность всех зависимостей, определяемых проектом, загружая их при необходимости; обычно это делается автоматически (например, при каждом изменении вектора `:dependencies`).

Примечание. Как видите, синтаксис определения зависимостей в Leiningen и Maven существенно отличается. Однако, он определяет одну и ту же информацию; так выглядит описание зависимости в Leiningen:

```
[org.clojure/clojure "1.3.0"]
```

которое в точности соответствует определению в Maven:

```
<dependency>
    <groupId>org.clojure</groupId>
    <artifactId>clojure</artifactId>
    <version>1.3.0</version>
</dependency>
```

Так как Leiningen предлагает более краткий синтаксис и более широко используется в сообществе Clojure (например, в большинстве проектов файлы README будут содержать определения векторов с зависимостями

---

[1] Например, `java -cp <path-to-uberjar> com.foo.MainClassName` или `java -cp <path-to-uberjar> clojure.main -m com.foo.namespace-with-main-fn`. Последняя особенно часто используется для проектов Clojure и не требует выполнения предварительной компиляции; подробности смотрите в документации для `clojure.main/main`.

в стиле Leiningen, вместо XML-фрагментов `<dependency>` в стиле Maven), далее в книге, для ссылки на библиотеку Clojure, используемую в примере, мы будем указывать соответствующую зависимость в нотации Leiningen.

Настройка предварительной компиляции

Помимо определения зависимостей и способов упаковки артефактов, одной из наиболее часто используемых операций в процессе сборки является предварительная компиляция исходных кодов Clojure. Однако, прежде чем продолжить, необходимо рассмотреть «за» и «против» предварительной компиляции. Загляните в раздел «Предварительная компиляция» выше, прежде чем принимать решение о необходимости или желательности предварительной компиляции для вашего проекта. Очень часто она не нужна.

В любом случае, оба инструмента, Leiningen и Maven, позволяют легко включать, отключать и настраивать предварительную компиляцию.

Leiningen. По умолчанию Leiningen не выполняет предварительную компиляцию файлов с исходным кодом. Чтобы включить компиляцию, необходимо добавить параметр `:aot` в настройки проекта, где этот параметр может иметь следующие значения:

- `:all` – требует от Leiningen выполнить компиляцию всех пространств имен, имеющихся в проекте;
- вектор пространств имен, определяющий, какие пространства имен в проекте должны компилироваться.

После добавления параметра `:aot` в настройки, запуск команды `lein compile` вызовет предварительную компиляцию пространств имен вашего проекта.

Maven. Предварительная компиляция в расширении `clojure-maven-plugin` включена по умолчанию, по крайней мере при использовании метода упаковки `clojure`, как показано в примере 8.3. Когда указывается этот метод упаковки, цель, отвечающая за компиляцию в расширении `clojure-maven-plugin`, связывается с фазой компиляции в Maven, в результате чего предварительная компиляция запускается после компиляции Java-кода, выполняемой по умолчанию.

Если вы используете иной метод упаковки, отличный от `clojure`, привязку к фазе компиляции необходимо настроить явно:

```
<plugin>
    <groupId>com.theoryinpractise</groupId>
    <artifactId>clojure-maven-plugin</artifactId>
```

```
    <version>1.3.8</version>
    <configuration>
        <warnOnReflection>true</warnOnReflection>
        <temporaryOutputDirectory>false</temporaryOutputDirectory>
    </configuration>

    <executions>
        <execution>
            <id>compile-clojure</id>
            <phase>compile</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Здесь также можно видеть два параметра, имеющие отношение к настройкам предварительной компиляции: `warnOnReflection` и `temporaryOutputDirectory`. Они управляют включением `*warn-on-reflection*` и сохранением файлов классов, сгенерированных в процессе компиляции, в каталог *classes* по умолчанию или во временный каталог, соответственно.

Предварительная компиляция в режиме проверки целостности. Даже если вы не предполагаете распространять скомпилированные файлы классов, включение предварительной компиляции в процедуру сборки (и игнорирование полученных файлов классов) может пригодиться для проверки целостности программного кода проекта. Так как для предварительной компиляции требуется загрузить компилируемый код, необходимо разрешить и загрузить все ссылки на библиотеки, пространства имен, переменные и Java-классы; если с какой-либо из этих ссылок возникнут проблемы, этап предварительной компиляции вскроет их[1].

Расширение `clojure-maven-plugin` предусматривает такую возможность; просто добавьте `<temporaryOutputDirectory>true</temporaryOutputDirectory>` в элемент `<configuration>` и результаты предварительной компиляции будут сохраняться во временном каталоге (который

[1] Те же проверки выполняются при загрузке кода в REPL, но вообще считается хорошей практикой гарантировать регулярное выполнение таких проверок, в процессе сборки, а не время от времени, в интерактивном сеансе REPL.

позднее будут удален). Это гарантирует предварительную оценку целостности кода и результаты компиляции не попадут в упакованный дистрибутив.

В настоящее время Leiningen не поддерживает простой способ выполнения предварительной компиляции и игнорирования получающихся файлов классов при упаковке артефактов проекта. Поэтому лучшее, что можно предложить – это вызвать команду `lein compile` (чтобы выполнить компиляцию) и затем вызвать `lein clean` перед запуском процедуры упаковки проекта.

Точно так же можно включить `*warn-on-reflection*` в ходе предварительной компиляции, что принудит компилятор Clojure выводить предупреждения для каждого встреченного обращения к механизму рефлексии или несоответствия типов аргументов[1]. Чтобы включить вывод этих предупреждений, добавьте `<warnOnReflection>true</warnOnReflection>` в настройки для расширения `clojure-maven-plugin` или `:warn-on-reflection true` – в файл *project.clj* для Leiningen.

Сборка гибридных проектов

Если потребуется добавить код на языке Clojure в существующий проект, вам, возможно, придется уделить некоторое внимание порядку, в каком компилируются различные части такого гибридного проекта[2]. Если Java-код ссылается на классы, сгенерированные формами определения типов языка Clojure в том же проекте, вам необходимо будет обеспечить предварительную компиляцию исходного кода на Clojure перед компиляцией исходных кодов на Java. Аналогично, если в коде на языке Clojure имеются ссылки на классы, определяемые в Java-коде, такой исходный код на Java должен компилироваться первым.

Примечание. Проблемы, характерные для гибридных проектов, могут вообще отсутствовать в проектах на Java, где не используется Clojure,

---

[1] См. раздел «Указание типов для производительности» в главе 9, где описываются особенности указания типов в операциях, и раздел «Ошибки и предупреждения, вызванные несоответствием типов» в главе 11, где подробнее рассказывается о предупреждениях, которые выводятся компилятором Clojure при несоответствии типов аргументов.

[2] Для простоты мы будем предполагать, что гибридный проект включает программный код только на двух языках, Clojure и Java. Аналогичный совет можно дать, тем, кто работает над проектами Clojure/JRuby или Scala/Clojure.

или если код на Java взаимодействует с кодом на Clojure, не ссылаясь на типы, определяемые в Clojure[1].

В любом случае этапы компиляции, присутствующие в процедуре сборки, необходимо упорядочить так, чтобы они отражали зависимости между кодом на разных языках внутри проекта. Оба инструмента, Leiningen и Maven, предусматривают такую возможность.

Maven. По умолчанию расширение `clojure-maven-plugin` включает компиляцию Java-кода перед предварительной компиляции кода на Clojure. Этот порядок можно изменить, связав цель компиляции для расширения `clojure-maven-plugin` с фазой, предшествующей фазе `compile`, такой как `process-resources`:

```
<plugin>
     <groupId>com.theoryinpractise</groupId>
     <artifactId>clojure-maven-plugin</artifactId>
     <version>1.3.8</version>
     <executions>
          <execution>
               <id>clojure-compile</id>
               <phase>process-resources</phase>
               <goals>
                    <goal>compile</goal>
               </goals>
          </execution>
     </executions>
</plugin>
```

Leiningen. В Leiningen по умолчанию предварительная компиляция Closure также выполняется после компиляции кода на Java, которая в свою очередь производится, если в вызове макроса `defproject` был определен слот `:java-source-path`, указывающий, где можно найти исходный код на Java.

Leiningen позволяет изменить этот порядок, но только через механизмы управления, доступные программно, а не через изменение настроек, как в Maven. Для этого, во-первых, необходимо оставить `:java-source-path` в вызове макроса `defproject` как есть; в противном случае порядок компиляции, принятый по умолчанию, не изменит-

[1] См. раздел «Использование Clojure из Java»в главе 9, где описывается, как использовать функциональность, реализованную на языке Clojure, из Java, не определяя в Clojure новые типы для использования их в Java.

ся. Далее нам нужно изменить поведение задания `compile` так, чтобы компилятор `javac` запускался после того, как будут выполнены действия, предусмотренные заданием по умолчанию. В состав Leiningen входит библиотека `robert-hooke`[1], обеспечивающая возможность реализовать это прямо в файле проекта *project.clj*:

```
(defproject com.clojurebook/lein-mixed-source "1.0.0"
  :dependencies [[org.clojure/clojure "1.3.0"]]
  :aot :all)

(require '(leiningen compile javac))                                    ❶

(add-hook #'leiningen.compile/compile                                   ❷
  (fn [compile project & args]                                          ❸
    (apply compile project args)                                        ❹
    (leiningen.javac/javac (assoc project :java-source-path "srcj"))))  ❺
```

❶ Сначала необходимо загрузить ключевые пространства имен Leiningen.

❷ Далее следует зарегистрировать точку входа в функцию-драйвер для задания `compile`; для этого в библиотеке `robert-hooke` имеется функция `add-hook`, которая сохраняет функцию драйвер в переменной `#'leiningen.compile/compile`.

❸ Функции-драйверу передаются: оригинальная функция `compile`, хранившаяся в переменной `#'leiningen.compile/compile`, текущие настройки проекта `project`, и все остальные аргументы, которые могли передаваться оригинальной функции `compile`. Когда `add-hook` вернет управление, наша функция-драйвер получит полный контроль над выполнением задания `compile`.

❹ Сначала мы вызываем оригинальную функцию `compile`, чтобы выполнить предварительную компиляцию кода на Clojure.

❺ Нам необходимо определить значение ключа `:java-source-path`, чтобы компилятор `javac` знал, где искать исходный код на языке Java. Обычно это значение включается в модель проекта (определяемую макросом `defproject` в начале файла *project.clj*); удаление этого параметра из настроек гарантирует, что `compile` не будет пытаться запустить `javac` до компиляции исходных текстов на языке Clojure.

---

[1] `robert-hooke` – это библиотека на языке Clojure, реализующая надмножество поддержки аспектно-ориентированного программирования (Aspect-Oriented Programming, AOP) в языке Clojure; ее применение для решения задач, которые в Java обычно решаются с помощью фреймворков AOP, описывается в разделе «Аспектно-ориентированное программирование», в главе 12.

Теперь при запуске команды `lein compile` будет вызываться наша функция-драйвер, выполняющая сначала предварительную компиляцию кода на языке Clojure, а затем запускающая компилятор `javac`, чтобы скомпилировать Java-код, находящийся в дереве каталогов с корнем в каталоге *srcj*.

Внимание. Однако вам придется организовать программный код внутри проекта так, чтобы полностью исключить чередование зависимостей в исходном коде. Под чередованием зависимостей понимаются такие ситуации, когда в одном проекте, например, имеется код на Java, использующий тип, определяемый в коде на Clojure, который в свою очередь реализует интерфейс, объявленный в Java. Подобные чередования можно разрешить (лучше всего, пожалуй, для этого подойдет Maven, учитывая, что этот инструмент позволяет указывать произвольное количество целей компиляции), но обычно их наличие является признаком недостаточно хорошо продуманной архитектуры проекта.

В заключение

Организация программного кода и программных проектов фактически является самостоятельной дисциплиной. Надеемся, мы смогли дать вам достаточно информации об организации и сборке проектов в стиле языка Clojure, чтобы вы могли продолжить дальнейшее изучение этой темы самостоятельно и применять полученные знания на практике.

Глава 9. Java и взаимодействие с JVM

Многие языки программирования предоставляют собственную среду времени выполнения. Популярными примерами такого подхода являются Python (CPython), Ruby (MRI) и Java (the JVM)[1]. В отличие от них, Clojure является *гостевым* (hosted) языком, в том смысле, что он предполагает использование существующей среды выполнения, в число которых входит и JVM[2]. Это означает, что вместо повторной реализации основных механизмов среды выполнения (таких как, механизмы сборки мусора, динамической компиляции (just-in-time compilation), многопоточного выполнения, поддержки графического контекста, и так далее) и разнообразных библиотек (от простых библиотек обработки строк до сложнейших, таких как библиотеки криптографических функций), Clojure просто использует все, что уже было сделано для JVM.

Помимо простой экономии времени на реализацию, использование зрелой среды выполнения несет дополнительные выгоды для программистов на Clojure.

- Основные механизмы JVM и экосистема библиотек поддерживаются серьезными техническими организациями. Это означает, что они обычно хорошо протестированы, широко используются и чрезвычайно оптимизированы, обеспечивая характеристики производительности, необходимые или желательные для большинства практиков.
- Опора на JVM означает возможность использования стандартных способов взаимодействий. Программы, написанные на одном языке, могут использовать функциональные возможности

---

[1] Начиная с версии Java 7, в JVM были внесены изменения, касающиеся исключительно языков, отличных от Java – таких как Clojure – что сделало эту виртуальную машину по-настоящему многоязычной.

[2] Существуют и другие реализации Clojure для выполнения в других виртуальных машинах; см. раздел «(dissoc Clojure 'JVM)» в главе 20.

библиотек, написанных на других языках, в единой среде выполнения, где интерфейсы и объектная модель Java является *универсальным средством общения* (lingua franca).

- Обширные сообщества пользователей Java и JVM гарантируют наличие широчайшего спектра библиотек для самых разных областей, изобилие документации и огромное количество разработчиков с глубочайшим пониманием платформы.
- Существуют разнообразные инструменты поддержки всех фаз разработки программного обеспечения, включая среды разработки, инструменты сборки, профилировщики, отладчики и управления проектами.

Для максимального использования возможностей Clojure совершенно необходимо иметь полное представление его отношений со средой выполнения, чтобы можно было использовать для пользы дела все самое лучшее, имеющееся в JVM, библиотеки, написанные для нее и все прежние наработки, которые могут у вас иметься.

JVM – основа Clojure

Можно было бы подумать, что будучи гостевым языком, Clojure некоторым образом отделен от JVM, подобно тому, как в некоторых средах выполнения сценариев строится своеобразная «песочница», покинуть которую не так-то просто. Однако это предположение весьма далеко от истины. Существует множество областей, где Clojure использует основные механизмы JVM. Назовем самые примечательные из них:

- строки Clojure являются строками Java;
- значение `nil` в Clojure – это значение `null` в Java;
- числа Clojure являются числами Java[1];
- регулярные выражения Clojure являются экземплярами класса `java.util.regex.Pattern`;
- структуры данных в Clojure реализуют соответствующие части интерфейсов `java.io.*` коллекций, обеспечивая доступ только для чтения; то есть, ассоциативные массивы в Clojure реализуют `java.util.Map`, векторы, последовательности и списки реализуют `java.util.List`, а множества реализуют `java.util.Set`;
- функции Clojure реализуют `java.lang.Runnable` и `java.util.concurrent.Callable`, существенно упрощая их интеграцию с су-

[1] Здесь есть некоторые тонкости; см. табл. 11.1.

ществующими библиотеками и фреймворками, ожидающими получить реализацию этих основных интерфейсов Java;
- оставляя в стороне синтаксис и абстракцию, вызовы функций в Clojure *являются* вызовами методов в Java; то есть, функции Clojure и их вызовы не влекут дополнительных накладных расходов;
- Clojure не является интерпретирующим языком – программный код всегда компилируется до уровня эффективного байткода JVM, даже в интерактивных окружениях, таких как REPL;
- вызовы методов Java API из Clojure семантически и фактически являются теми же операциями, что и вызовы тех же API из Java:
- функции Clojure компилируются в классы;
- формы `defrecord` и `deftype` Clojure компилируются в классы Java, содержащие обычные поля;
- определения протоколов, выполненные с помощью `defprotocol`, генерируют соответствующие интерфейсы Java.

Такая глубокая интеграция означает, что для использования библиотек Java из языка Clojure[1] (и наоборот) обычно не требуется применять специальные обертки, преобразования или другие ухищрения, и это не влечет снижение производительности, в сравнении с эквивалентным кодом на Java.

Кроме того, благодаря глубине стандартной библиотеки Java и огромной численности сообщества, сплотившегося вокруг JVM, весьма маловероятно, что вы не сможете найти нужную вам библиотеку для JVM (чаще встречается обратная ситуация, когда существует несколько конкурирующих между собой библиотек). Как правило, это весьма благодатная почва для пришедших из других окружений, где часто бывает необходимо повторно реализовать те или иные функциональные возможности для конкретного языка.

Использование классов, методов и полей Java

В языке Clojure предусмотрены некоторые простейшие формы взаимодействий с классами, методами и полями виртуальной машины; благодаря этому программный код на языке Clojure, выполняю-

---

[1] Который, напомню, сам является библиотекой Java; см. врезку «Clojure – «всего лишь» одна из зависимостей», в главе 8.

щий операции с Java-библиотеками выглядит весьма естественно и оказывается более кратким, чем эквивалентный код на языке Java.

Таблица 9.1. Формы взаимодействий в Clojure и их эквиваленты на Java[1]

Операция	Форма Clojure	Эквивалент Java
Создание экземпляра класса `ClassName`	(ClassName.) (ClassName. arg1 arg2 ...)	new ClassName() new ClassName(arg1, arg2, ...)
Вызов метода экземпляра `object`	(.methodName object) (.methodName object arg1 arg2 ...)	object.methodName() object.methodName(arg1, arg2, ...)
Вызов статического метода `staticMethod` класса `ClassName`	(ClassName/staticMethod) (ClassName/staticMethod arg1 arg2 ...)	ClassName.staticMethod() ClassName. staticMethod(arg1, arg2, ...)
Доступ к значению статического поля `FIELD` в классе `ClassName`	ClassName/FIELD	ClassName.FIELD
Ссылка на класс `ClassName`	ClassName	ClassName.class
Доступ к значению поля экземпляра `field` в объекте `object`	(.field object)	object.field
Запись значения 5 в поле экземпляра `field` объекта `object`	(set! (.fieldName object) 5)	object.fieldName = 5

В примерах 9.1 и 9.2 приводятся короткие сеансы работы в REPL, где демонстрируется применение всех форм взаимодействий с некоторыми основными классами из стандартной библиотеки Java:

Пример 9.1. Извлечение веб-страницы с помощью библиотек Java

```
(import 'java.net.URL)                    ❶
;= java.net.URL
(def cnn (URL. "http://cnn.com"))         ❷
;= #'user/cnn
```

[1] Мы немного погрешили против истины: в действительности существует всего две основные формы взаимодействий с виртуальной машиной, «точка» (`.`) и `new`. Первая обеспечивает возможность вызова методов и обращения к полям, а вторая – возможность вызова конструкторов. Кроме `set!`, все остальные формы в табл. 9.1 являются различными вариантами использования «точки» (`.`) и `new`. одробнее этот синтаксический сахар описывается в разделе «Взаимодействие с Java: . и new», в главе 1.

```
(.getHost cnn)                                    ❸
;= "cnn.com"
(slurp cnn)                                       ❹
;= "<html lang=\"en\"><head><title>CNN.com........."
```

❶ Макрос `import` импортирует указанный класс в текущее пространство имен; в данном случае стандартный класс `URL`.

❷ Здесь выполняется создание экземпляра класса `URL` на основе единственного аргумента, и результат сохраняется в переменной. Данная операция эквивалентна вызову конструктора в Java `new URL("http://cnn.com")`.

❸ В результате вызова метода `getHost` возвращается именно то, что мы и ожидали; эта операция эквивалентна вызову метода `url.getHost()` в Java.

❹ В Clojure имеется функция `slurp`, возвращающая строковое представление множества объектов различных типов, включая `java.io.File`, `java.net.Socket`, массивов байтов и многих других[1]. Здесь функция `slurp` получает экземпляр класса `URL` и возвращает содержимое страницы по указанному адресу URL в виде строки.

Доступ к статическим методам и полям выполняется так же просто:

Пример 9.2. Доступ к статическим методам и полям Java

```
Double/MAX_VALUE
;= 1.7976931348623157E308
(Double/parseDouble "3.141592653589793")
;= 3.141592653589793
```

Важно отметить, что все эти формы взаимодействий полностью совместимы с понятием позиции функции[2]: выполняемая «операция» всегда определяется первым символом в любой форме. То есть, в дополнение к функциональной совместимости с вызовами «родных» методов и конструкторов Java, формы взаимодействий полностью соответствуют синтаксису языка Clojure. Они читаются

[1] В действительности функция `slurp` опирается на функцию `clojure.java.io/reader`, возвращающую `java.io.Reader` для всех этих объектов разных типов. В нашем примере выполняются различные операции взаимодействий с Java: чтобы открыть сокет `java.net.Socket` с указанным URL и получить `java.io.InputStream` из этого сокета, который затем обертывается экземпляром `Reader`. После того, как функция `slurp` получит его, она просто читает текстовые данные из экземпляра `Reader`.

[2] Подробнее о позиции функции рассказывается в разделе «Выражения, операторы, синтаксис и очередность», в главе 1.

и пишутся так же естественно, как и обычный код, вызывающий обычные функции Clojure, и допускают возможность использовать их в сочетании с конструкциями языка Clojure, такими как семейство потоковых макросов ->. Например, ниже приводится идиоматическая функция на языке Clojure, которая принимает строку с представлением десятичного числа и возвращает строку с шестнадцатеричным представлением:

```
(defn decimal-to-hex
  [x]
  (-> x
    Integer/parseInt
    (Integer/toString 16)
    .toUpperCase))
;= #'user/decimal-to-hex
(decimal-to-hex "255")
;= "FF"
```

Макрос -> передает результаты выполнения каждой предыдущей формы в первом аргументе следующей формы (попутно преобразуя «голые» символы в списки из одного элемента) и возвращает результат выполнения последней формы. Таким образом, тело функции `decimal-to-hex` можно представить, как `(.toUpperCase (Integer/toString (Integer/parseInt x) 16))`; или, на языке Java:

```
public String stringToHex (String x) {
    return Integer.toString(Integer.parseInt(x), 16).toUpperCase();
}
```

На наш взгляд, применение макроса -> (или подобного ему, такого как ->>[1]) позволяет получить более читаемый код (его можно рассматривать как линейный конвейер, не вынуждающий скользить взглядом взад-вперед по выражению, так как все операции расположены в порядке выполнения слева направо), полностью совместимый с формами взаимодействий, благодаря последовательному использованию позиций функций.

Доступ к полям объектов. Общедоступные поля редко встречаются в Java API, а изменяемые общедоступные поля – особенно. Но, как бы то ни было, Clojure обеспечивает простой доступ к ним,

[1] Подробное описание макросов -> и ->> см. в разделе «Подробнее: -> и ->>», в главе 5.

и простую возможность изменения их значений с помощью специальной формы `set!`[1]:

Пример 9.3. Чтение и запись полей Java-объектов

```
(import 'java.awt.Point)
;= java.awt.Point
(def pt (Point. 5 10))
;= #'user/pt
(.x pt)
;= 5
(set! (.x pt) -42)
;= -42
(.x pt)
;= -42
```

Обратите внимание, что в форме `set!` используется тот же синтаксис – `(.x pt)` – для ссылки на модифицируемое поле, что и в операции извлечения значения поля.

> **Внимание. Изменяемость в Java (лазейка или удобная возможность).** В то время, как Clojure поощряет использование неизменяемых значений (как обсуждалось в главе 2), этого нельзя сказать об основных языках программирования, использующих виртуальную машину JVM вместе с Clojure. Коллекции в Java поощряют изменение состояния объекта. Так как в Clojure имеется возможность использовать библиотеки, классы и объекты Java, вы можете оперировать изменяемым состоянием без лишних церемоний. Это может быть одновременно и удобной возможностью (особенно при использовании функциональности, доступной только в виде библиотеки Java) или лазейкой (при неосторожном обращении с изменяемыми объектами Java). В любом случае, будьте внимательны при работе с объектами Java из Clojure, особенно если вы понимаете и полагаетесь на особенности Clojure.

Удобные утилиты взаимодействий

Работа с библиотеками Java в значительной степени связана с использованием форм взаимодействий, перечисленных в табл. 9.1. Однако, существует несколько удобных и часто используемых утилит, о которых вам следует знать:

---

[1] Форма `set!` используется также в ряде других ситуаций, позволяя напрямую модифицировать изменяемые ячейки; подробности см. в разделе «Специализированная операция set!», в главе 1.

`class`

Возвращает `java.lang.Class` своего аргумента; например `(class "foo") => java.lang.String`

`instance?`

Функция-предикат, возвращающая `true`, если второй аргумент является экземпляром класса, имя которого указано в первом аргументе, например `(instance? String "foo") => true`

`doto`

Макрос, вызывающий вложенные формы и передающий свой первый аргумент в виде первого аргумента каждой форме.

Назначение и порядок использования `class` и `instance?` должны быть очевидны, но макрос `doto` требует некоторых пояснений.

Многие классы в мире Java являются изменяемыми и часто требуют инициализации за пределами их конструкторов. Хорошим примером может служить класс `java.util.ArrayList`, экземпляры которого часто приходится заполнять некоторыми начальными данными:

```
ArrayList list = new ArrayList();
list.add(1);
list.add(2);
list.add(3);
```

Конечно, в языке Clojure имеются литералы, представляющие его собственные векторы[1], но иногда существующие библиотеки Java API могут *требовать* передачи именно экземпляра `ArrayList` (а не `java.util.List`, когда можно было бы передать непосредственно вектор Clojure). В таких случаях можно с помощью формы `let` создать `ArrayList`, заполнить соответствующими данными и вернуть ссылку на `ArrayList`:

Пример 9.4. Заполнение массива ArrayList «вручную»

```
(let [alist (ArrayList.)]
  (.add alist 1)
```

[1] Обратите внимание, что векторы в Clojure не имеют ничего общего с классом `java.util.Vector`. Полное обсуждение структур данных в языке Clojure можно найти в главе 3.

```
  (.add alist 2)
  (.add alist 3)
  alist)
```

Этот прием работает, но код получился такой же многословный, как и в Java. Макрос `doto` дает более удобную возможность[1]:

Пример 9.5. Использование doto для заполнения ArrayList

```
(doto (ArrayList.)
  (.add 1)
  (.add 2)
  (.add 3))
```

Этот пример делает в точности то же самое, что и пример 9.4, но он стал заметно короче. Выгоды от использования макроса `doto` становятся еще более очевидными при необходимости определить длинную последовательность операций над единственным объектом. Например:

Пример 9.6. Использование doto для управления контекстом java.awt.Graphics2D

```
(doto graphics
  (.setBackground Color/white)
  (.setColor Color/black)
  (.scale 2 2)
  (.clearRect 0 0 500 500)
  (.drawRect 100 100 300 300))
```

Здесь к объекту `Graphics2D` применяется длинная последовательность операций, в результате которых в графическом контексте, возвращаемом самим макросом `doto`, рисуется черный квадрат внутри белого квадрата.

Главной идиомой языка Clojure является работа со значениями, а не процедурное изменение фиксированных ссылок. Предоставляя возможность обозначать границы процедурной инициализации или других операций с побочными эффектами, и неявно возвращая объект этих операций, который далее в программе на Clojure можно

---

[1] Конечно, `ArrayList` очень удобно использовать для демонстрации возможностей макроса `doto`, однако его можно заполнить более простым способом – использовав коллекцию Clojure: `(ArrayList. [1 2 3])`.

интерпретировать как обычное значение, `doto` помогает объединить операции взаимодействий и объекты с поддержкой состояния с этой идиомой.

Исключения и обработка ошибок

Clojure использует механизм исключений JVM *целиком*[1], со знакомой идиомой `try/catch/finally/throw`. Таким образом, схема обработки ошибок в Clojure будет знакома всем, кто когда-либо работал с языками, использующими эту идиому, включая Java, Ruby, Python и многие другие.

С учетом вышесказанного, блоки кода в примере 9.7 являются эквивалентными, с точки зрения обработки ошибок:

Пример 9.7. Парсинг строки в целое число на Java, Ruby и Clojure с обработкой ошибок

```
// Java
public static Integer asInt (String s) {
     try {
          return Integer.parseInt(s);
     } catch (NumberFormatException e) {
          e.printStackTrace();
          return null;
     } finally {
          System.out.println("Attempted to parse as integer: " + s);
     }
}

# Ruby
def as_int (s)
     begin
          return Integer(s)
     rescue Exception => e
          puts e.backtrace
     ensure
          puts "Attempted to parse as integer: " + s
```

[1] Существует множество библиотек Clojure, реализующих расширенную обработку ошибок, с более широкими возможностями, чем описываются здесь; в частности, библиотека Slingshot предоставляет намного более сложный механизм моделирования и обработки ошибок: https://github.com/scgilardi/slingshot.

```
    end
end

; Clojure
(defn as-int
  [s]
  (try
    (Integer/parseInt s)
    (catch NumberFormatException e
      (.printStackTrace e))
    (finally
      (println "Attempted to parse as integer: " s))))
```

Операция парсинга целых чисел является весьма обыденной, но на ее примере отчетливо видно, что семантика принцип действия форм `try`, `catch` и `finally` в Clojure полностью соответствует аналогичным операторам в Java и соответствующим операторам в Ruby (`begin`, `rescue`, `ensure`):

`try`
: Определяет границы формы обработки исключений. Может содержать произвольное количество форм `catch` и необязательных форм `finally`, которым может предшествовать произвольное количество выражений, представляющих «удачно выполняющийся» код[1]. В случае успешного выполнения, результатом всей формы `try` является результат последнего выражения предшествующего какой-либо форме `catch` или `finally`.

`catch`
: Определяет некоторый код для выполнения в случае появления исключения указанного типа (например, `java.lang.Throwable` или любого из его подклассов) в ходе выполнения основного тела формы `try`. Возникшее исключение будет привязано к имени, следующему за типом исключения. Последнее выражение в активированной форме `catch` будет определять

[1] Следует отметить, что формы `catch` и `finally` должны включаться внутрь формы `try`; это делает более явной семантику взаимоотношений между `try` (или, например, `begin` в Ruby) и `catch` с `finally`, последняя из которых почти во всех языках синтаксически интерпретируется как инструкция одного уровня с `try`.

результат вмещающей формы `try`. Указать можно любое количество форм `catch`.

`finally`
: Определяет программный код для выполнения непосредственно перед тем как поток управления покинет форму `try`, независимо от причин, повлекших выход (то есть, выражения в форме `finally` будут выполнены, даже если какой-то код в теле формы `try` сгенерировал не перехваченное исключение). Форма `finally` не влияет на результат формы `try`; по этой причине единственная польза от нее заключается в побочных эффектах. Единственная проблема – если исключение будет сгенерировано в теле формы `finally`, оно замаскирует любое другое исключение, сгенерированное в соответствующей форме `try` (как и в языке Java).

Это – базовые механизмы обработки ошибок в JVM (и, соответственно, в Clojure).

Возбуждение исключений. Код на языке Clojure может генерировать исключения с помощью формы `throw`, которая является точным аналогом инструкции `throw` в Java и инструкции `raise` в Ruby и Python:

```
(throw (IllegalStateException. "I don't know what to do!"))
;= #<IllegalStateException java.lang.IllegalStateException: I don't know
;=   what to do!>
```

С синтаксической точки зрения в Python инструкции `raise` можно передать любой класс, а в Ruby – даже строку, однако в Clojure форме `throw` может передаваться только экземпляр класса, наследующего `java.lang.Throwable` или один из его подклассов:

```
(throw "foo")
;= #<ClassCastException java.lang.ClassCastException:
;= java.lang.String cannot be cast to java.lang.Throwable>
```

Повторное использование существующих типов исключений. Одной из идиом, которую часто можно заметить в программном коде на Java, но очень редко – в коде на Clojure, является определение новых типов исключений. Идиоматический код Clojure, когда генерирует исключения, использует исключения стандартных типов.

В стандартной библиотеке Java определено более 350 типов исключений; хотя такое и *возможно*, однако вам крайне редко придется сталкиваться с ситуациями, когда ни один из типов исключений не подойдет для описания ошибочного состояния. Можно с уверенностью сказать, что в 90 процентах случаев с успехом можно использовать «базовые» типы исключений:

```
java.lang.IllegalArgumentException;
java.lang.UnsupportedOperationException;
java.lang.IllegalStateException;
java.io.IOException.
```

Это – общее правило, но иногда без собственных типов исключений не обойтись. Если вы окажетесь в такой ситуации, вам может пригодиться пример определения типа исключения в разделе «Определение собственного типа исключения» (ниже).

Отказ от контролируемых исключений

Несмотря на то, что Clojure базируется на JVM, он не унаследовал *контролируемые исключения* (checked exceptions) Java. Это такие типы исключений, которые могут объявляться, как генерируемые Java-методами; компилятор Java будет требовать от кода, вызывающего эти методы, чтобы он перехватывал и обрабатывал контролируемые исключения или объявлял, что сам может генерировать эти исключения. Контролируемые исключения уже долгое время являются источником споров в экосистеме Java[1], однако почти все согласны, что контролируемые исключения делают программный код слишком подробным и слишком сложным в сопровождении.

К счастью Clojure не подвержен ограничениям, связанным с необходимостью обрабатывать контролируемые исключения, объявляемые методами Java, которые вызываются из Clojure, так как контролируемые исключения являются продуктом компилятора Java и просто не существуют в JVM[2]. Поэтому можно (например) создавать временные файлы, используя метод, который объявлен, как способный генерировать контролируемое исключение `java.io.IOException`:

---

[1] Отличное исследование на эту тему: http://www.mindview.net/Etc/Discussions/CheckedExceptions.

[2] Во многом похоже на то, как стираются обобщенные классы (generics) Java.

```
(File/createTempFile "clojureTempFile" ".txt")
```

...без использования выражения `try/catch` и без объявлений `throw`, чтобы показать, что код может генерировать исключение типа `IOException`. Это позволяет писать более краткий код на Clojure, вызывающий методы с объявлениями контролируемых исключений, в сравнении с эквивалентным кодом на Java.

with-open, прощай finally

Чаще всего форма `finally` используется, чтобы гарантировать корректное управление ресурсами. Например, ниже демонстрируется статический метод на языке Java, добавляющий некоторый текст в конец файла:

Пример 9.8. Добавление в конец файла на Java с управлением ресурсами вручную с применением finally

```
public static void appendTo (File f, String text) throws IOException {
     Writer w = null;
     try {
          w = new OutputStreamWriter(new FileOutputStream(f, true), "UTF-8");
          w.write(text);
          w.flush();
     } finally {
          if (w != null) w.close();
     }
}
```

Обратите внимание, что здесь в блоке `finally` мы по условию закрываем экземпляр `Writer`, открытый в предшествующем блоке `try`. Этот шаблон (в котором так легко ошибиться) является постоянным источником ошибок во многих программах, работающих с файлами, сокетами, базами данных и другими ресурсами, требующими явного управления. По этой причине в Java 7 была введена инструкция `try-with-resources`[1], которая автоматически закрывает, например, дескрипторы файлов при выходе из области видимости и очень напоминает инструкцию `with` в Python:

---

[1] Если вы еще не знакомы с ней, примеры ее использования можно найти по адресу: http://download.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html.

Пример 9.9. Добавление в конец файла на Java 7 с использованием try-with-resources

```
public static void appendTo (File f, String text) throws IOException {
     try (Writer w = new OutputStreamWriter(new FileOutputStream(f, true),
                                                    "UTF-8")) {
          w.write(text);
          w.flush();
     }
}
```

В языке Clojure имеется эквивалентная форма `with-open`, гарантирующая освобождение ресурсов, перед тем как поток управления покинет область видимости[1]. Ниже приводится идиоматическая реализация метода `appendTo` на языке Clojure, где используется форма `with-open`, гарантирующая корректное закрытие экземпляра `Writer`:

Пример 9.10. Добавление в конец файла на Clojure с автоматическим управлением ресурсом с помощью with-open

```
(require '[clojure.java.io :as io])

(defn append-to
  [f text]
  (with-open [w (io/writer f :append true)]     ❶
    (doto w (.write text) .flush)))
```

❶ Мы могли бы вручную создать `OutputStreamWriter` и `FileInputStream`, но вспомогательная функция `writer` гораздо удобнее в этом отношении.

При необходимости с формой `with-open` можно связать несколько ресурсов[2], каждый из которых будет автоматически закрыт перед тем

---

[1] Мы не можем упустить возможность упомянуть, что форма `with-open` реализована как макрос из девяти строк, который легко можно было бы написать самостоятельно, что очень ярко подчеркивает преимущество Clojure перед другими языками, не имеющими макросов, где вы вынуждены ждать, пока разработчики внесут желаемые улучшения. Подробнее о макросах и их возможностях рассказывается в главе 5.

[2] Здесь можно использовать любые классы, имеющие метод `close`, соответствующий методу `close`, определяемому интерфейсом `java.lang.Closeable`, но, так как Clojure является динамическим языком, от ресурсов, используемых в форме `with-open`, не требуется, чтобы они реализовали этот интерфейс. Во многих других динамических языках это называется «утиной типизацией» (duck typing).

как поток управления покинет форму with-open. Ниже представлена упрощенная реализация функции копирования, демонстрирующая такую возможность:

Пример 9.11. Добавление в конец файла на Clojure с автоматическим управлением ресурсом с помощью with-open

```
(defn copy-files
  [from to]
  (with-open [in (FileInputStream. from)
              out (FileOutputStream. to)]
    (loop [buf (make-array Byte/TYPE 1024)]
      (let [len (.read in buf)]
        (when (pos? len)
          (.write out buf 0 len)
          (recur buf))))))
```

Это – отличный пример использования with-open, но гораздо предпочтительнее было бы использовать функцию copy (и другие функции ввода/вывода) из пространства имен clojure.java.io[1].

Указание типов для производительности

В некоторых примерах программного кода можно заметить, что для ссылки на имена классов Java используется синтаксис, такой как ^String в следующем фрагменте:

```
(defn length-of
  [^String text]
  (.length text))
```

Синтаксис ^ClassName определяет *указание на тип* (type hint), явно подсказывающее компилятору Clojure тип выражения, значения переменной или именованной привязки.

Обычно синтаксическая конструкция указания на тип преобразуется механизмом чтения Clojure в метаданные значения, следующего за указанием. То есть, указание типа в привязке ^String text будет преобразовано механизмом чтения в эквивалентное выраже-

[1] http://clojure.github.com/clojure/clojure.java.io-api.html.

ние `^{:tag String} text`, которое будет интерпретироваться как символ `text` с метаданными `{:tag String}`. Для определения типа в Clojure используется терм «tag».

Эти указания на тип используются только компилятором, чтобы избежать обращений к механизму рефлексии при взаимодействиях с JVM; ни для чего другого они не используются. Поэтому, если в программном коде не используются методs взаимодействий с JVM, указания на тип можно не включать:

```
(defn silly-function
  [v]
  (nil? v))
```

Любое указание на тип для привязки `v` здесь просто не будет использоваться.

Внимание. Указания на типы аргументов функции или возвращаемого значения не являются объявлениями сигнатуры: они не оказывают влияния на типы значений, которые может принимать или возвращать функция. Они используются исключительно, чтобы обеспечить включение прямых вызовов Java-методов и операций доступа к полям Java-объектов в скомпилированный код, вместо того, чтобы использовать намного более медлительный механизм рефлексии для поиска методов или полей во время выполнения. То есть, если указание на тип производится не в форме, взаимодействующей с JVM, оно фактически никак не учитывается. Например, в следующей функции указывается, что ее аргумент имеет тип `java.util.List`, однако она может принять аргумент любого другого типа:

```
(defn accepts-anything
  [^java.util.List x]
  x)
;= #'user/accepts-anything
(accepts-anything (java.util.ArrayList.))
;= #<ArrayList []>
(accepts-anything 5)
;= 5
(accepts-anything false)
;= false
```

В отличие от этого механизма, объявления сигнатур функций, поддерживаемые в языке Clojure, позволяют объявлять только простые типы аргументов и возвращаемых значений. Подробнее об объявлении простых типов будет рассказываться в разделе «Объявление функций, принимающих и возвращающих значения простых типов», в главе 11.

Исключение из программы вызовов механизма рефлексии является ключом к высокой производительности в коде, выполняющем массивные вычисления. Практически, чтобы полностью исключить использование механизма рефлексии, требуется совсем небольшое количество указаний на тип, поскольку компилятор Clojure поддерживает автоматическое определение типов, опираясь на известные типы литералов, вызовы конструкторов и типы значений, возвращаемых методами.

Чтобы продемонстрировать это, добавим указание на тип в некоторый код, с целью оптимизировать его. Ниже приводится функция, возвращающая строку, в которой все знаки приведены к верхнему регистру:

Пример 9.12. Функция преобразования регистра без указания на тип

```
(defn capitalize
  [s]
  (-> s
    (.charAt 0)
    Character/toUpperCase
    (str (.substring s 1))))
```

Эта реализация прекрасно работает, однако ее производительность можно немного увеличить:

Пример 9.13. Хронометраж преобразования строки «foo» 100 000 раз

```
(time (doseq [s (repeat 100000 "foo")]
        (capitalize s)))
; "Elapsed time: 5040.218 msecs"
```

В подобных случаях, когда есть подозрение, что добавление указаний на типы могут дать некоторый прирост производительности, включайте предупреждения об использовании механизма рефлексии в компиляторе Clojure; это поможет определить, действительно ли производятся вызовы методов механизма рефлексии[1]:

---

[1] Прием, представленный здесь, удобно использовать в оболочке REPL, однако те же предупреждения можно получить на этапе сборки проекта, с дополнительным указанием номеров строк. За подробностями обращайтесь к разделу «Предварительная компиляция в режиме проверки целостности» (глава 8).

Пример 9.14. Получение предупреждений об использовании механизма рефлексии для функции преобразования регистра, представленной в примере 9.12

```
(set! *warn-on-reflection* true)
;= true
(defn capitalize
  [s]
  (-> s
    (.charAt 0)
    Character/toUpperCase
    (str (.substring s 1))))
; Reflection warning, NO_SOURCE_PATH:27 - call to charAt can't be resolved.
; Reflection warning, NO_SOURCE_PATH:29 - call to toUpperCase can't be
; resolved.
; Reflection warning, NO_SOURCE_PATH:29 - call to substring can't be
; resolved.
;= #'user/capitalize
```

Здесь видно, что функция производит три операции взаимодействия с JVM, которые выполняются с применением механизма рефлексии. Эту проблему можно решить, добавив единственное указание на тип (обратите внимание на дополнительное объявление `^String`):

```
(defn fast-capitalize
  [^String s]
  (-> s
    (.charAt 0)
    Character/toUpperCase
    (str (.substring s 1))))
```

Это объявление устраняет все три обращения к механизму рефлексии. Как может единственное указание на тип оказывать влияние на все три вызова? Это обусловлено тем, что в игру вступил механизм определения типов компилятора Clojure:

1. В `let`-привязке для имени `s` явно указан тип `String`. Поэтому...
2. ...вызов `.charAt` может быть скомпилирован в непосредственный вызов `String.charAt`. Компилятор знает, что этот метод возвращает значение типа `char`, благодаря чему...
3. ...он может корректно выбрать версию `Character.toUpperCase`, возвращающую значение типа `char` (вместо перегруженной версии, возвращающей значение типа `int`).

4. Наконец, компилятор снова использует указание на тип `String` для `s` и выбрать метод `String.substring` при компиляции вызова метода `.substring`.

Посмотрим, как это сказалось на производительности переделанной версии функции `fast-capitalize`:

```
(time (doseq [s (repeat 100000 "foo")]
          (fast-capitalize s)))
; "Elapsed time: 154.889 msecs"
```

Продолжительность вычислений упала с 5 секунд до 0.155 секунды. Неплохо!

Указание на тип можно добавить в любое выражение. Взгляните на функцию, которая обращается к механизму рефлексии, чтобы вызвать метод `String.split`:

```
(defn split-name
  [user]
    (zipmap [:first :last]
(.split (:name user) " ")))
;= #'user/split-name
; Reflection warning, NO_SOURCE_PATH:3 - call to split can't be resolved.
(split-name {:name "Chas Emerick"})
;= {:last "Emerick", :first "Chas"}
```

Если бы тип можно было указывать только для имен в `let`-привязках, нам пришлось бы добавить форму `let`, чтобы создать промежуточное значение для `(:name user)` и указать его тип:

```
(defn split-name
  [user]
  (let [^String full-name (:name user)]
    (zipmap [:first :last]
      (.split full-name " "))))
;= #'user/split-name
```

Обращения к механизму рефлексии исчезли, но функция получилась слишком длинной. К счастью, существует возможность непосредственно указать тип выражения `(:name user)`:

```
(defn split-name
  [user]
```

```
  (zipmap [:first :last]
    (.split ^String (:name user) " ")))
;= #'user/split-name
```

Никаких обращений к механизму рефлексии и никаких лишних операций. Аналогично можно указать тип значения, возвращаемого функцией, чтобы в вызывающем ее программном коде не приходилось указывать тип получаемого от нее значения при взаимодействиях с JVM:

```
(defn file-extension
  [^java.io.File f]
  (-> (re-seq #"\.(.+)" (.getName f))
    first
    second))

(.toUpperCase (file-extension (java.io.File. "image.png")))
; Reflection warning, NO_SOURCE_PATH:1 - reference to field toUpperCase
;  can't be resolved.
;= "PNG"
```

Указание на тип возвращаемого значения прилагается *к вектору аргументов*:

```
(defn file-extension
  ^String [^java.io.File f]
  (-> (re-seq #"\.(.+)" (.getName f))
    first
    second))

(.toUpperCase (file-extension (java.io.File. "image.png")))
;= "PNG"
```

Наконец, указание на тип можно добавлять к именам переменных, чтобы указать, значения каких типов они содержат:

```
(def a "image.png")
;= #'user/a
(java.io.File. a)
; Reflection warning, NO_SOURCE_PATH:1 - call to java.io.File ctor can't be
; resolved.
;= #<File image.png>
(def ^String a "image.png")
```

```
;= #'user/a
(java.io.File. a)
;= #<File image.png>
```

Массивы

До появления API `java.util.Collections` в Java 1.2, массивы[1] были одним из немногих способов хранения больших пакетов объектов. В настоящее время массивы редко используются для хранения объектов, но они остаются важным инструментом для работы с большими наборами данных числовых и других простых типов. В любом случае, в Clojure предусматривается возможность работы с массивами Java, однако это – одна из немногих областей, где Clojure является более многословным, чем Java, особенно в сравнении со специальным синтаксисом в последнем.

Таблица 9.2. Сравнение операций с массивами

Операция	Выражение на языке Clojure	Эквивалент на Java
Создание массива из коллекции	(into-array [«a» «b» «c»])	(String[])coll.toArray(new String[list.size()]);
Создание пустого массива	(make-array Integer 10 100)	new Integer[10][100]
Создание пустого массива значений простого типа `long`	(long-array 10) (make-array Long/TYPE 10)	new long[10]
Доступ к значению в массиве	(aget some-array 4)	some_array[4]
Изменение значения в массиве[a]	(aset some-array 4 «foo») (aset ^ints int-array 4 5)	some_array[4] = 5.6

[a] При изменении значений в массивах простых типов, в игру вступает механизм определения типов в Clojure. Подробности см. в разделе «Используйте простые массивы осмысленно», в главе 11.

Если программный код на Clojure просто получает массивы объектов, возможно, возвращаемые существующими Java API, тогда нет необходимости явно преобразовывать их в списки или другие коллекции. Массивы поддерживаются абстракцией последователь-

[1] Не путайте массивы в Clojure/Java с типом `Array` в Ruby. Тип `Array` в Ruby больше напоминает тип `Vector` в Java.

ностей языка Clojure[1], поэтому их можно использовать как любые другие упорядоченные коллекции:

```
(map #(Character/toUpperCase %) (.toCharArray "Clojure"))
;= (\C \L \O \J \U \R \E)
```

Однако в Clojure имеется специальная поддержка массивов значений простых типов. Мы будем рассматривать эту поддержку в разделе «Используйте простые массивы осмысленно», в главе 11.

Определение классов и реализация интерфейсов

Возможность вызывать методы Java и создавать экземпляры классов хороша сама по себе, но часто бывает необходимо определять собственные классы, реализовать интерфейсы и наследовать существующие классы. В Clojure имеется множество средств определения классов, каждое из которых предлагает сплав различных возможностей для разных ситуаций.

Таблица 9.3. Сравнение ключевых особенностей форм в языке Clojure для определения классов Java[2]

	proxy	gen-class	reify	deftype	defrecord
Возвращает экземпляр анонимного класса	✓		✓		
Определяет именованный класс		✓		✓	✓
Позволяет наследовать существующий базовый класс	✓	✓			
Позволяет определять новые поля				✓	✓
Предоставляет реализацию по умолчанию методов `Object.equals` и `Object.hashcode`, а также различных интерфейсов Clojure					✓

[1] В частности функция `seq`, лежащая в основе всех операций с последовательностями в Clojure, получив массив вернет соответствующую ему последовательность.

[2] Если вы знаете, что потребуется определить новый тип на языке Clojure, но не уверены, какую форму следует выбрать, обращайтесь к диаграмме в главе 18, предназначенной специально для этой цели.

Все эти формы можно использовать для определения классов и реализации интерфейсов. Некоторые из этих форм – `deftype`, `defrecord` и `reify` – играют особую роль в Clojure, которая никак не связана с классами и интерфейсами Java. Поэтому их мы рассматривали отдельно в главе 6.

Две другие – `proxy` и `gen-class` – предназначены исключительно для поддержки классов и интерфейсов Java и будут рассматриваться здесь.

Экземпляры анонимных классов: proxy

`proxy` создает экземпляр анонимного класса, реализующего произвольное количество интерфейсов Java и/или один конкретный базовый класс[1]. Этот анонимный класс создается только один раз, на этапе компиляции, на основе указанного базового класса и интерфейсов. Стоимость каждого вызова `proxy` во время выполнения обусловливается всего лишь стоимостью единственного вызова конструктора сгенерированного класса. Это делает форму `proxy` эквивалентом определения и создания экземпляра анонимного класса в Java.

Чтобы продемонстрировать порядок использования proxy, рассмотрим, как с ее помощью можно реализовать простой LRU-кеш (основанный на алгоритме с вытеснением по наибольшему времени неиспользования – Least Recently Used)[2] из строительных блоков, имеющихся в стандартной библиотеке Java.

В JDK имеется реализация ассоциативных массивов на основе хешей, `java.util.LinkedHashMap`, которая может служить основой для реализации простейшего LRU-кеша: она поддерживает возможность обхода элементов в порядке последнего обращения и определяет метод `removeEldestEntry(Map.Entry<K,V>)`, который можно переопределить в подклассе, чтобы сообщить базовому классу `LinkedHashMap`, какой элемент является старейшим (в смысле времени создания или последнего обращения) и может быть удален.

Реализация такого кеша с помощью `proxy` позволит нам познакомиться со всеми особенностями этой формы:

---

[1] Если не требуется наследовать существующий конкретный класс, предпочтительнее использовать форму `reify`, которая описывается в разделе «Определение анонимных типов с помощью reify», в главе 6.

[2] Если вы не знакомы с алгоритмом кеширования LRU, обращайтесь к статье: http://ru.wikipedia.org/wiki/Алгоритмы_кэширования.

Пример 9.15. Реализация простейшего LRU-кеша с использованием LinkedHashMap и proxy

```
(defn lru-cache                                                    ❶
  [max-size]                                                       ❷
  (proxy [java.util.LinkedHashMap] [16 0.75 true]                  ❸
    (removeEldestEntry [entry]                                     ❹
      (> (count this) max-size))))                                 ❺
```

❶ Сначала определим фабричную функцию, чтобы можно было получать экземпляры этого кеша по мере необходимости.

❷ Методы, реализованные с помощью `proxy`, образуют замыкания, поэтому любые значения, привязанные в области видимости, где используется форма `proxy`, остаются доступными внутри этих методов. В данном случае наш метод `removeEldestEntry` замыкает аргумент `max-size` фабричной функции. Сравнивая это значение с размером ассоциативного массива, мы определяем необходимость удаления элемента.

❸ Форма `proxy` требует передачи ей двух векторов в первых двух аргументах:

1. Имя суперкласса и/или реализуемых интерфейсов; Обратите внимание, что имя суперкласса должно следовать первым.
2. Аргументы для передачи конструктору суперкласса. В данном примере мы передаем конструктору `LinkedHashMap` значения по умолчанию, инициализирующие размер ассоциативного массива (`16`), коэффициент загрузки (`0.75`) и логическое значение `true`, указывающее, что порядок обхода элементов определяется в порядке последнего доступа к ним, а не в порядке их добавления. Именно эта особенность превращает экземпляр `LinkedHashMap` в действующий LRU-кеш. Обратите внимание, что в вектор с аргументами конструктора можно было бы использовать не только литералы, но и любые выражения.

❹ Форма `proxy` не требует передачи первого аргумента `this` в методы, реализованные с ее помощью (в отличие от других форм определения классов в языке Clojure, `reify`, `defrecord` и `deftype`). Она неявно связывает аргумент `this` с экземпляром `proxy`...

❺ ...который мы используем для проверки размера ассоциативного массива. Если он окажется больше замкнутого значения `max-size`, тогда мы возвращаем `true`, показывая тем самым, что дольше всех не использовавшийся элемент должен быть вытеснен из кеша. Конечно, для реализации политики вытеснения можно использовать любой критерий, однако размер ассоциативного массива ничуть не хуже других и часто используется для этих целей.

Реализации методов в форме `proxy` могут следовать в любом порядке.

Попробуем воспользоваться нашим простым LRU-кешем в оболочке REPL и убедимся, что он работает, как ожидалось:

```
(def cache (doto (lru-cache 5)                                    ❶
                 (.put :a :b)))
;= #'user/cache
cache
;= #<LinkedHashMap$0 {:a=:b}>
(doseq [[k v] (partition 2 (range 500))]                          ❷
  (get cache :a)                                                  ❸
  (.put cache k v))                                               ❹
;= nil
cache
;= #<LinkedHashMap$0 {492=493, 494=495, 496=497, :a=:b, 498=499}>  ❺
```

❶ Сначала создадим новый кеш, воспользовавшись фабричной функцией `lru-cache` из примера 9.15. Добавим один элемент, связывающий между собой `:a` и `:b`[1]. Мы будем часто обращаться к этому элементу, чтобы он никогда не был вытеснен из кеша, как наиболее давно не использовавшийся.

❷ Соберем последовательность из 250 двухэлементных списков с помощью `partition`, содержащих монотонно увеличивающиеся целые числа из указанного диапазона `range`[2]. Каждый из этих двухэлементных списков деструктурируется[3] в привязки ключ/значение с помощью форму деструктуризации `[k v]`.

❸ Перед добавлением нового элемента в кеш, выполняется обращение к элементу, соответствующему ключу `:a`. Это позволяет предотвратить вытеснение данного элемента из кеша, благодаря поддерживанию его позиции в `LinkedHashMap`, поэтому он никогда не будет предложен методу `removeEldestEntry` для вытеснения.

❹ Каждую пару ключ/значение, полученную из последовательности, созданной в ❷, мы помещаем в кеш. Как только будет превышен предел `max-size`, равный 5, каждый вызов `.put` должен приводить к вытеснению из кеша элемента, который не использовался дольше всех.

❺ После завершения большого количества вызовов `.put`, проверяется состояние кеша...

---

[1] `:a` и `:b` являются литералами ключевых слов в Clojure. Ключевые слова описываются в разделе «Ключевые слова (keywords)», в главе 1.

[2] Примером такой последовательности могла бы служить более короткая последовательность, чтобы ее можно было вывести на экран: `(partition 2 (range 10)) => ((0 1) (2 3) (4 5) (6 7) (8 9))`.

[3] Прочитать о формах деструктуризации можно в разделе «Деструктуризация (let, часть 2)», в главе 1.

...и, как и следовало ожидать, кеш содержит всего пять элементов, и среди них – элемент `[:a :b]`; они ни разу не был предложен для вытеснения методу `removeEldestEntry`, реализованному с помощью `proxy`, и потому остался в кеше. Только элемент `[498 499]` оказался более свежим, чем наш элемент `[:a :b]`, так как последней операцией с ассоциативным массивом было добавление элемента `[498 499]`.

Определение именованных классов

Форма `proxy` позволяет определять анонимные в коде на языке Clojure классы во время выполнения, однако часто бывает необходимо определить статический, именованный Java-класс для передачи пользователям Java[1]. В Clojure имеется три формы создания именованных классов, каждая из которых имеет свои достоинства и недостатки.

Вместе с протоколами, `deftype` и `defrecord` представляют принципиальный подход Clojure к моделированию данных и предметных областей. Как следствие, они не поддерживают некоторые сложные аспекты модели объектов в Java, чтобы уступить дорогу идиомам Clojure. С другой стороны, для достижения максимальной эффективности, `deftype` и `defrecord` структурно напоминают «обычные» Java-классы: они позволяют определять новые поля (необязательно простых типов) а реализации их методов встраиваются (inlined) в файлы классов, которые они генерируют, что делает их такими же эффективными, как и классы, реализованные на языке Java. Можно сказать, что `deftype` и `defrecord` используют «самые лучшие стороны» объектной модели Java. Хотя они могут и используются для взаимодействия с Java, когда это возможно[2], тем не менее, они имеют более широкое предназначение в Clojure; по этой причине мы обсуждали их отдельно в главе 6.

Форма `gen-class`, напротив, предоставляет более полноценную поддержку объектной модели Java, – включая определение стати-

---

[1] Этими пользователями могут быть программисты на Java, если вы распространяете свой код на Clojure в виде библиотеки, или библиотеки для JVM, фреймворки и серверы, требующие, чтобы в файлах конфигурации указывались имена статических реализаций классов. Типичным примером могут служить контейнеры сервлетов; в их файлах *web.xml*, как описывается в примере 17.1, требуется указывать именованные классы сервлетов.

[2] Как будет показано далее в этой главе; см. разделы «Реализация конечных точек веб-службы JAX-RS» и «Использование классов, созданных с помощью deftype и defrecord» ниже.

ческих методов, наследование конкретных базовых классов, а также определение нескольких конструкторов и новых методов экземпляров – но созданные с ее помощью классы структурно отличаются от типичных классов на языке Java, так как содержат автоматически сгенерированные методы, которые во время выполнения вызывают обычные функции Clojure.

gen-class

Форма `gen-class` позволяет определять Java-классы, реализации методов которых опираются на обычные функции Clojure. Она предназначена исключительно для использования в контексте взаимодействий и поддерживает большую часть модели объектов в Java, что позволяет обеспечивать соответствие требования фреймоврков и библиотек, за редким исключениями. Она позволяет:

- сгенерировать Java-класс с любым именем и в любом пакете;
- унаследовать существующий базовый класс и получить доступ к защищенным (protected) полям;
- реализовать в классе произвольное количество интерфейсов;
- определять любое количество конструкторов;
- определять статические и дополнительные методы экземпляров, сверх тех, что уже имеются в базовом суперклассе и реализуемых интерфейсах;
- генерировать статические фабричные функции;
- генерировать статический метод `main` для классов, которые должны быть доступны из командной строки.

Внимание. `gen-class` – единственная форма в языке Clojure, которая должна быть скомпилирована предварительно. Без этого формы `gen-class` превращаются в пустые операции, так как `gen-class` не определяет класс во время выполнения, подобно другим формам определения классов в Clojure. На этапе предварительной компиляции формы `gen-class` генерируют файлы классов Java, которые могут вкладываться в .jar-файлы и использоваться другими библиотеками и программами на Java.
Подробнее о предварительной компиляции рассказывается в разделе «Предварительная компиляция» в главе 8. Все примеры в этой главе, где используется форма `gen-class`, предполагают, что они будут скомпилированы предварительно.

Для полноценного описания всех возможностей `gen-class` потребовалась бы отдельная глава. Поэтому мы представим лишь пару примеров использования `gen-class`, которые помогут вам начать самим разбираться с особенностями ее работы. Ниже представлено

пространство имен, содержащее упрощенную реализацию операции изменения размеров изображения, которую можно упаковать в автономный выполняемый *.jar*-файл, благодаря использованию определения `gen-class`:

Пример 9.16. Реализация статических методов и утилиты командной строки с помощью gen-class

```
(ns com.clojurebook.imaging
  (:use [clojure.java.io :only (file)])
  (:import (java.awt Image Graphics2D)
           javax.imageio.ImageIO
           java.awt.image.BufferedImage
           java.awt.geom.AffineTransform))

(defn load-image
  [file-or-path]
  (-> file-or-path file ImageIO/read))

(defn resize-image
  ^BufferedImage [^Image original factor]
  (let [scaled (BufferedImage. (* factor (.getWidth original))
                               (* factor (.getHeight original))
                               (.getType original))]
    (.drawImage ^Graphics2D (.getGraphics scaled)
                original
                (AffineTransform/getScaleInstance factor factor)
                nil)
    scaled))

(gen-class
  :name ResizeImage                                                    ❶
  :main true                                                           ❷
  :methods [^:static [resizeFile [String String double] void]          ❸
            ^:static [resize [java.awt.Image double] java.awt.image.BufferedImage]])

(def ^:private -resize resize-image)                                   ❹

(defn- -resizeFile
  [path outpath factor]
  (ImageIO/write (-> path load-image (resize-image factor))
                 "png"
                 (file outpath)))

(defn -main                                                            ❺
```

```
[& [path outpath factor]]
(when-not (and path outpath factor)
  (println "Usage: java -jar example-uberjar.jar ResizeImage [INFILE]
    [OUTFILE] [SCALE]")
  (System/exit 1))
(-resizeFile path outpath (Double/parseDouble factor)))
```

❶ По умолчанию форма `gen-class` генерирует класс с именем, совпадающим с названием пространства имен, в котором она находится. В данном случае, название пространства имен (`com.clojurebook.imaging`) является идиоматичным для Clojure, но оно не соответствует общепринятой практике в Java и слишком длинное, чтобы использовать его в командной строке. Поэтому мы определили имя класса как `ResizeImage`.

❷ Нам хотелось бы иметь возможность использовать этот класс из командной строки, поэтому мы включили в форму `gen-class` параметр `main`. Он обеспечит создание метода `public static void main String(args[])`, который будет вызывать функцию `-main` в данном пространстве имен.

❸ Здесь определяются два метода для генерируемого класса. Определения записываются в форме: `[имяМетода [типы параметров] типВозвращаемогоЗначения]`. Оба метода являются статическими, о чем свидетельствуют метаданные `^:static`, присоединяемые к каждому вектору с сигнатурой метода.

❹ По умолчанию форма `gen-class` ищет функции, реализующие методы, в том же пространстве имен, где находится сама форма `gen-class`, и с теми же именами, что и в объявлениях методов, но с префиксом `-`[1]. Статический метод `resize` объявлен для удобства – он имеет ту же семантику и сигнатуру, что и функция `resize-image`. То есть мы просто создали псевдоним для функции `resize-image` в новой переменной с именем `-resize`, которой метод `resize` будет делегировать выполнение операции. Для метода `resizeFile` мы определили отдельную реализацию.

❺ Метод `main` передает аргументы командной строки методу `resizeFile`, а в случае их отсутствия – выводит информацию о порядке использования.

После компиляции этого пространства имен мы получим файл класса `ResizeImage`, который можно будет запустить из командной строки или использовать в Java-приложении, вызывая его статические методы. Например, допустим, что в текущем каталоге имеется изображение *clojure.png*, представленное на рис. 9.1.

---

[1] При желании можно использовать другой префикс, указав его в виде строки в слоте `:prefix` формы `gen-class`.

Рис. 9.1. Исходное изображение clojure.png

Мы можем запустить нашу утилиту `ResizeImage` из командной строки:

```
java -cp gen-class-1.0.0-standalone.jar ResizeImage clojure.png resized.png 0.5
```

и получить уменьшенное изображение, как показано на рис. 9.2.

Рис. 9.2. Изображение после изменения размеров с помощью утилиты ResizeImage

Статические методы класса `ResizeImage` доступны также из программного кода на Java:

```
ResizeImage.resizeFile("clojure.png", "resized.png", 0.5);
```

Важно отметить, что форма `gen-class` не требует что-либо менять в программном коде Clojure. Пространство имен `com.clojurebook.imaging` является прекрасным (хотя и небольшим) образцом Clojure API, идиоматичным во всех отношениях – мы лишь добавили в него форму `gen-class`, чтобы открыть доступ к функциональности этого пространства имен из Java.

Определение собственных типов исключений. Как уже говорилось в разделе «Повторное использование существующих типов исключений» (выше), для Clojure типично повторно использовать типы исключений, имеющиеся в стандартной библиотеке Java или в сторонних библиотеках, которые могли бы использоваться в приложении. Однако иногда требуется использовать специализированные типы исключений, особенно если вы сотрудничаете с коллегами, которые пишут на Java и привыкли создавать свои типы исключений для каждой ошибки.

Рассмотрим нестандартный тип исключения, несущего в себе ассоциативный массив, вдобавок к типичным для Java значениям типа `String` и `Throwable`:

Пример 9.17. Определение собственного типа исключений с помощью gen-class

```
(ns com.clojurebook.CustomException
  (:gen-class
     :extends RuntimeException                                          ❶
     :implements [clojure.lang.IDeref]                                  ❷
     :constructors {[java.util.Map String] [String]                     ❸
                    [java.util.Map String Throwable] [String Throwable]}
     :init init
     :state info                                                        ❹
     :methods [[getInfo [] java.util.Map]                               ❺
               [addInfo [Object Object] void]]))

(import 'com.clojurebook.CustomException)

(defn- -init                                                            ❻
  ([info message]
    [[message] (atom (into {} info))])                                  ❼
  ([info message ex]
    [[message ex] (atom (into {} info))]))

(defn- -deref                                                           ❽
```

```
  [^CustomException this]
  @(.info this))

(defn- -getInfo
  [this]
  @this)

(defn- -addInfo
  [^CustomException this key value]
  (swap! (.info this) assoc key value))
```

❶ Наш тип исключения наследует `java.lang.Exception`...

❷ ...и реализует один из интерфейсов Clojure, `java.lang.IDeref`, обеспечивая тем самым поддержку абстракции `deref`, описанную в примечании, в разделе «delay», в главе 4. Благодаря этому программный код на языке Clojure может использовать `deref` и `@` для извлечения ассоциативного массива из исключений этого типа.

❸ Здесь мы определяем пару конструкторов. Ассоциативный массив (в этой его части) указывает, что конструктор `CustomException(java.util.Map, String)` будет вызывать конструктор `Exception(String)` суперкласса. Значения, которые фактически будут переданы конструктору суперкласса, определяются функцией, идентифицируемой слотом `:init`.

❹ Наш тип исключений будет иметь единственное финальное (final) поле с именем `info`. Как будет использоваться это поле, мы увидим чуть ниже.

❺ Здесь определяются два метода, `getInfo` и `addInfo`. Их удобство для Java API нашего нестандартного исключения, будет продемонстрировано чуть ниже.

❻ Форма `gen-class` генерирует конструкторы, опираясь на указанные нами сигнатуры. Эти конструкторы будут вызывать функцию `:init` с теми же аргументами, какие получат сами. В этой функции можно выполнить те же самые операции по инициализации, что и в обычном конструкторе Java.

❼ Функция `:init` всегда должна возвращать вектор с двумя элементами: первый – вектор аргументов для передачи конструктору суперкласса, а второй – значение для записи в финальное поле `:state`. В поле `:state` мы будем сохранять атом, содержащий ассоциативный массив `info`, а так как для координации изменений в ассоциативном массиве `info` используется атом, мы копируем исходный ассоциативный массив (возможно изменяемый) в неизменяемый массив Clojure.

❽ Функции `-deref` и `-addInfo` (реализующие методы `deref` и `addInfo`) демонстрируют, как можно оперировать атомом, хранящимся в финальном поле `info` класса `CustomException`.

В противоположность примеру 9.16, где так же использовалась форма `gen-class`, пространство имен `com.clojurebook.CustomException` было создано исключительно с целью определить класс `CustomException`. В таких случаях настройку формы `gen-class` можно «встраивать» непосредственно в объявление пространства имен, позволив классу унаследовать имя от пространства имен.

А теперь посмотрим, как можно использовать этот класс из Clojure, на примере нескольких простых функций, аналоги которых легко можно найти во многих крупных приложениях:

```
(import 'com.clojurebook.CustomException)
;= nil
(defn perform-operation
  [& [job priority :as args]]
  (throw (CustomException. {:arguments args} "Operation failed")))          ❶
;= #'user/perform-operation
(defn run-batch-job
  [customer-id]
  (doseq [[job priority] {:send-newsletter :low
                          :verify-billings :critical
                          :run-payroll :medium}]
    (try
      (perform-operation job priority)
      (catch CustomException e                                              ❷
        (swap! (.info e) merge {:customer-id customer-id
                                :timestamp (System/currentTimeMillis)})
        (throw e)))))
;= #'user/run-batch-job
(try
  (run-batch-job 89045)
  (catch CustomException e
    (println "Error!" (.getMessage) @e)))                                   ❸
; Error! Operation failed {:timestamp 1309935234556, :customer-id 89045,
;                          :arguments (:verify-billings :critical)}
;= nil
```

❶ Функция `perform-operation` возбуждает новое исключение `CustomException`, помещая в ассоциативный массив `info` аргументы, переданные ей.

❷ Любая функция верхнего уровня (здесь `run-batch-job`) находящаяся в цепочке вызовов, сможет перехватить исключение `CustomException` и добавить новые данные в его ассоциативный массив `info`. Поскольку выполнение протекает в программном коде Clojure, нет необходимо-

сти использовать метод `addInfo`, созданный в форме `gen-class`, – можно просто извлечь атом из поля `info` и добавить в ассоциативный массив дополнительную информацию: числовой идентификатор (ID) клиента и время, когда возникло исключение.

❸ Функция верхнего уровня может просто перехватывать и обрабатывать исключения `CustomException`; то есть вместо добавления информации она может:

- разыменовать экземпляр исключения (с помощью макроса `@`), вызвав его метод `deref`, и получить накопленный ассоциативный массив с информацией;
- с помощью метода `.getMessage` извлечь оригинальное сообщение, полученное исключением в момент его создания; обратите внимание, что мы не определяли этот метод – классы, созданные с помощью формы `gen-class`, наследуют методы базовых классов, как обычные классы Java.

Возможность передавать произвольные данные вместе с исключением, как в данном примере, может оказаться весьма мощной особенностью. При желании вместе с исключениями можно передавать даже функции, которые другой программный код мог бы вызывать, чтобы повторить операцию, возможно, с другими аргументами[1].

Использование нового типа исключений в Java так же не вызывает сложностей:

```
import com.clojurebook.CustomException;
import clojure.lang.PersistentHashMap;

public class BatchJob {
    private static void performOperation (String jobId, String priority) {
        throw new CustomException(PersistentHashMap.create("jobId", jobId,
                "priority", priority), "Operation failed");
    }
```

[1] Этот прием можно использовать для реализации механизма *перезапуска*. Возможность перезапуска является важнейшей особенностью *систем слежения за состоянием*, обобщенных средств обработки ошибок на основе исключений, которые имеются в Smalltalk и в некоторых диалектах Lisp. такие системы позволяют любому коду, столкнувшемуся с исключительной ситуацией, обеспечить возможность перезапуска для программного кода более высокого уровня. И снова мы рекомендуем обратить внимание на библиотеку Slingshot, если у вас появится желание поэкспериментировать с более гибким механизмом обработки ошибок: https://github.com/scgilardi/slingshot.

```
    private static void runBatchJob (int customerId) {
        try {
            performOperation("verify-billings", "critical");
        } catch (CustomException e) {
            e.addInfo("customer-id", customerId);
            e.addInfo("timestamp", System.currentTimeMillis());
            throw e;
        }
    }

    public static void main (String[] args) {
        try {
            runBatchJob(89045);
        } catch (CustomException e) {
            System.out.println("Error! " + e.getMessage() + " " +
                               e.getInfo());
        }
    }
}
```

Единственное отличие от использования `CustomException` в Clojure состоит в том, что для выполнения операции `swap!` над атомом, хранящим поле `info` класса исключения, в Java предпочтительнее использовать метод `.addInfo`, а вместо метода `.deref` лучше использовать метод `.getInfo`, так как последний возвращает `java.util.Map`.

Аннотации

Аннотации в Java являются своего рода статическими метаданными, которые можно присоединять к объявлениям классов, методов и полей. Эти метаданные могут использоваться во время компиляции различными средствами создания программного кода и другими инструментами, применяемыми на этапе компиляции, или во время выполнения, с помощью механизма рефлексии языка Java. Поддержка аннотаций впервые появилась в Java 5 с целью дать пользователям библиотек и фреймворков возможность декларативно определять поведение и семантику рядом с затрагиваемыми программными элементами. Этим аннотации отличаются от XML-файлов и других механизмов конфигурирования, отделяющими метаданные от элементов, которые они описывают. В настоящее время аннотации получили широкое распространение во многих окружения Java, поэтому для Clojure очень важно иметь возможность прозрачно работать с такими контекстами.

Аннотации для интеграции

Аннотации – одна из особенностей JVM, которые заставляют съеживаться программистов на Clojure. Отчасти потому, что они могут служить источником немалых сложностей, даже в Java. Однако в действительности, аннотации Java при всей своей сложности делают не так много, в сравнении с комбинацией поддержки метаданных макросов и возможности компиляции во время выполнения в языке Clojure.

Поэтому, несмотря на великое множество библиотек и приложений на языке Clojure, благополучно использующих практически все аспекты взаимодействий с JVM, которыми обладает Clojure, лишь немногие используют аннотации Java и то, если только это действительно необходимо для интеграции.

Создание аннотированных тестов для JUnit

Clojure интерпретирует аннотации, присоединяемые к любым формам создания классов, как аннотации к этим классам, методам или полям. Рассмотрим пример добавления к методу аннотации `org.junit.Test` из популярного фреймворка тестирования JUnit (http://junit.org), с целью указать, какие методы, объявленные в форме `gen-class`, должны интерпретироваться как тесты.

Пример 9.18. Использование аннотаций JUnit для объявления тестовых методов

```
(ns com.clojurebook.annotations.junit
  (:import (org.junit Test Assert))
  (:gen-class
    :name com.clojurebook.annotations.JUnitTest
    :methods [[^{org.junit.Test true} simpleTest [] void]                    ❶
              [^{org.junit.Test {:timeout 2000}} timeoutTest [] void]        ❷
              [^{org.junit.Test {:expected NullPointerException}}            ❸
                badException [] void]]))

(defn -simpleTest
  [this]
  (Assert/assertEquals (class this) com.clojurebook.annotations.JUnitTest))

(defn -badException
  [this]
  (Integer/parseInt (System/getProperty "nonexistent")))                     ❹

(defn -timeoutTest
  [this]
  (Thread/sleep 10000))                                                      ❺
```

❶ Аннотированный метод simpleTest класса, создаваемого формой gen-class. Конструкция ^{org.junit.Test true} эквивалентна простой аннотации @org.junit.Test в Java без параметров.
❷ Здесь определяется значение 2000 миллисекунд для поля timeout аннотации org.junit.Test, применяемой к методу timeoutTest. Это объявление эквивалентно аннотации @org.junit.Test(timeout=2000) в Java.
❸ Аналогично мы определяем, что метод badException должен возбуждать исключение NullPointerException при вызове, указав класс в поле expected аннотации org.junit.Test. Это объявление эквивалентно аннотации @org.junit.Test(expected=Null PointerException) в Java.
❹ Наша реализация метода badException пытается преобразовать содержимое несуществующего системного свойства в целое число, в результате чего генерируется исключение NumberFormatException, а не NullPointerException, которое мы указали в поле expected аннотации.
❺ Наша реализация метода timeoutTest приостанавливает выполнение на 10 секунд, то есть на более долгий промежуток времени, чем 2 секунды, как указано в поле timeout аннотации.

В результате предварительной компиляции этого пространства имен будет создан класс com.clojurebook.annotations.JUnit Test, который можно передать инструменту запуска тестирования в JUnit. Один тест будет выполняться успешно (проверка в методе simple-Test всегда будет давать истинный результат), но другие два будут терпеть неудачу из-за несоответствия информации, переданной в метаданных аннотаций. В ходе тестирования The JUnit выведет следующие строки:

```
There were 2 failures:
1) timeoutTest(com.clojurebook.annotations.JUnitTest)
java.lang.Exception: test timed out after 2000 milliseconds
2) throwsWrongException(com.clojurebook.annotations.JUnitTest)
java.lang.Exception: Unexpected exception,
expected<java.lang.NullPointerException> but was <java.lang.
NumberFormatException>
```

Аннотации, указанные нами в объявлении :methods внутри формы gen-class благополучно определили критерии тестирования получившегося класса и были переданы механизму тестирования JUnit.

Реализация конечных точек веб-службы JAX-RS

JAX-RS – один из самых популярных стандартов создания веб-служб в мире Java. Он определяет API на основе аннотаций для

создания служб в стиле REST с использованием стандартных классов Java. Контейнеры, реализующие этот стандарт, используют аннотации для обнаружения классов, соответствующих адресам URL запросов, определения методов классов, соответствующих HTTP-методам запросов, и определения, например, содержимого заголовков `Content-Type` в HTTP-ответах.

Давайте определим класс *ресурса* JAX-RS на языке Clojure. Для этого можно было бы снова использовать форму `gen-class`, но на этот раз мы воспользуемся макросом `deftype`, чтобы продемонстрировать некоторые возможности поддержки аннотаций в Clojure[1]:

Пример 9.19. Веб-служба, реализованная с применением аннотаций JAX-RS

```
(ns com.clojurebook.annotations.jaxrs
  (:import (javax.ws.rs Path PathParam Produces GET)))

(definterface Greeting                                              ❶
  (greet [^String visitor-name]))

(deftype ^{Path "/greet/{visitorname}"} GreetingResource []         ❷
  Greeting
  (^{GET true                                                       ❸
     Produces ["text/plain"]}
    greet
    [this ^{PathParam "visitorname"} visitor-name]                  ❹
    (format "Hello %s!" visitor-name)))
```

❶ Форма `definterface` используется для определения интерфейса с единственным методом `Greeting` для нашего класса, определяемого с помощью `deftype`. Он принимает единственный строковый аргумент.

❷ С помощью формы `deftype` определяется класс, отмеченный аннотацией `Path`, имеющей значение `"/greet/{visitorname}"`; это означает, что любой запрос к контейнеру JAX-RS, соответствующий данному шаблону URL, будет передан для обработки нашему классу `GreetingResource`.

❸ Реализация нашего метода `greet` отмечена двумя аннотациями: `GET`, сообщающей, что этот метод должен вызываться для обработки `GET`-запросов, и `Produces`, определяющей значение для заголовка `Content-`

[1] Подробное описание макроса `deftype` можно найти в разделе «Определение собственных типов», в главе 6.

Type ответа, возвращаемого контейнером JAX-RS. В данном случае метод greet возвращает простую строку, поэтому наиболее подходящим будет значение "text/plain".

❹ Шаблон URL, который мы определили в аннотации Path класса, содержит единственный параметр visitorname. Добавив аннотацию PathParam с тем же именем параметра, что использовался в шаблоне URL, мы указываем, что этот параметр URL должен передаваться методу в виде аргумента visitor-name.

После компиляции класс GreetingResource можно развернуть в любом контейнере JAX-RS. Запустить такой контейнер можно в оболочке REPL, воспользовавшись встроенным веб-сервером Grizzly:

```
(com.sun.jersey.api.container.grizzly.GrizzlyWebContainerFactory/create
  "http://localhost:8080/"
  {"com.sun.jersey.config.property.packages"
   "com.clojurebook.annotations.jaxrs"})
```

Эта команда запустит экземпляр Grizzly, который будет принимать запросы по адресу localhost:8080. Веб-сервер выполнит поиск обработчиков ресурсов в пакете com.clojurebook.annotations.jaxrs, найдет наш класс GreetingResource и будет использовать его, как кандидата на обработку запросов. Если теперь обратиться по адресу http://localhost:8080/application.wadl, мы получим описание WADL контейнера JAX-RS, где можно увидеть наш URL ресурса, параметр visitorname и тип содержимого ответа text/plain:

```
% curl http://localhost:8080/application.wadl
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://research.sun.com/wadl/2006/10">
    <doc xmlns:jersey="http://jersey.java.net/"
         jersey:generatedBy="Jersey: 1.8 06/24/2011 12:17 PM"/>
    <resources base="http://localhost:8080/">
        <resource path="/greet/{visitorname}">
            <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   type="xs:string" style="template" name="visitorname"/>
            <method name="GET" id="greet">
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
        </resource>
    </resources>
</application>
```

Послав запрос GET с любым URL, имеющим вид http://localhost:8080/greet/<некоторое-имя>, мы получим результат, генерируемый нашим аннотированным классом ресурса JAX-RS:

```
% curl http://localhost:8080/greet/Jose
Hello Jose!
```

Использование Clojure из Java

Предположим на мгновение, что вам потребовалось использовать библиотеку на Clojure из Java и эта библиотека не определяет никаких типов или классов[1]. Чтобы воспользоваться библиотекой, необходимо получить доступ к функциям на языке Clojure и постоянным значениям, определяемым в пространствах имен. К счастью, сделать это из Java достаточно просто:

1. Загрузите требуемый код на Clojure. Для этого необходимо вызвать стандартные функции require, use или load из пространства имен clojure.core.
2. Получите ссылки на переменные, соответствующие каждой функции или значению, которые вы собираетесь использовать.
3. Вызывайте функции и используйте значения, необходимые приложению.

Для демонстрации взаимодействий Java→Clojure нам потребуются две переменные, одна функция и некоторое значение. Значение будет извлекаться из простого пространства имен Clojure:

Пример 9.20. Простое пространство имен Clojure

```
(ns com.clojurebook.histogram)

(def keywords (map keyword
                   '(a c a d b c a d c d k d a b b b c d e e e f a a a a)))
```

В качестве функции будет использоваться функция frequencies из пространства имен clojure.core. Она принимает любое значение, которое можно преобразовать в последовательность, и возвращает ассоциативный массив с элементами последовательности и счет-

[1] Приемы, представленные здесь, могут применяться к любым другим языкам JVM; просто переведите программный код в примере 9.21 на желаемый язык.

чиками – количествами вхождений каждого элемента в последовательности[1].

Ниже представлен Java-класс, использующий функцию `frequencies` со значением `keywords` и многое другое.

Пример 9.21. Использование кода на языке Clojure в примере 9.20 из Java

```
package com.clojurebook;

import java.util.ArrayList;
import java.util.Map;

import clojure.lang.IFn;
import clojure.lang.Keyword;
import clojure.lang.RT;
import clojure.lang.Symbol;
import clojure.lang.Var;

public class JavaClojureInterop {
     private static IFn requireFn = RT.var("clojure.core", "require").fn();  ❶
     private static IFn randIntFn = RT.var("clojure.core", "rand-int").fn();
     static {
          requireFn.invoke(Symbol.intern("com.clojurebook.histogram"));       ❷
     }

     private static IFn frequencies =
          RT.var("clojure.core", "frequencies").fn();                         ❸
     private static Object keywords = RT.var("com.clojurebook.histogram",     ❹
               "keywords").deref();

     @SuppressWarnings({ "unchecked", "rawtypes" })
     public static void main(String[] args) {
          Map<Keyword, Integer> sampleHistogram =
               (Map<Keyword, Integer>)frequencies.invoke(keywords);           ❺
          System.out.println("Number of :a keywords in sample histogram: " +
                    sampleHistogram.get(Keyword.intern("a")));                ❻
          System.out.println("Complete sample histogram: " + sampleHistogram);
          System.out.println();

          System.out.println(
```

[1] Результатом фактически является гистограмма. Узнать, что такое гистограмма, можно по адресу: http://ru.wikipedia.org/wiki/Гистограмма.

```
            "Histogram of chars in 'I left my heart in san fransisco': " +
            frequencies.invoke(
                "I left my heart in San Fransisco".toLowerCase()));      ❼
        System.out.println();

        ArrayList randomInts = new ArrayList();
        for (int i = 0; i < 500; i++) randomInts.add(randIntFn.invoke(10));
        System.out.println("Histogram of 500 random ints [0,10): " +
                frequencies.invoke(randomInts));                         ❽
    }
}
```

❶ Для начала ms извлекаем пару функций из стандартной библиотеки, `require` и `rand-int`, для последующего использования. Обратите внимание, что для этой цели используется метод `fn()`; он возвращает функции Clojure (каждая из которых реализует интерфейс `IFn`). Единственное отличие между `fn()` и `deref()` состоит в том, что первый автоматически выполняет приведение к типу `IFn`.

❷ Прежде чем пытаться обратиться к какому-либо пространству имен Clojure кроме `clojure.core`, его необходимо загрузить; здесь выполняется загрузка пространства имен с помощью нашей ссылки `requireFn` типа `IFn`. Эта строка является точным эквивалентом выражения `(require 'com.clojurebook.histogram)` в языке Clojure.

❸ Здесь мы получаем ссылку на переменную `clojure.core/frequencies`; эта строка эквивалентна выражению `#'clojure.core/frequencies` в языке Clojure.

❹ Здесь выполняется разыменование (`deref`) значения переменной `sample-values`, которое представляет собой последовательность ключевых слов, из примера 9.20.

❺ Мы вызываем функцию `frequencies` и передаем ей наши исходные, используя для этого метод `invoke()` интерфейса `IFn`. Обратите внимание, что этот метод (как и `Var.deref()`) возвращает `Object`; Clojure – динамический язык, поэтому переменные могут хранить значения *любых типов*. В Clojure эта особенность обрабатывается естественным образом, но требует некоторых усилий в таких статически типизированных окружениях, как Java: в данном случае нам известно, что функция `frequencies` возвращает `Map`, и что данные, поставляемые ей, имеют тип `Keyword`, поэтому мы спокойно можем выполнить приведение типа результата функции `frequencies` к `Map<Keyword, Number>`.

❻ Значения, возвращаемые из Clojure, являются обычными Java-объектами и мы можем использовать их в этом качестве. К ассоциативному массиву `Map` ключей `Keyword` и чисел `Number` можно обращаться точно так же, как если бы он был возвращен Java-методом. Здесь мы извле-

каем ключевое слово `:a`, чтобы увидеть, сколько раз оно встретилось в исходной последовательности.

❼❽ Благодаря обобщенной интерпретации различных конкретных типов в языке Clojure, мы также можем передать функции `frequencies` созданный в Java список `List`, строку `String`, и получить корректный результат.

Если скомпилировать и запустить этот Java-класс, он выведет следующие результаты:

```
% java -cp target/java-clojure-interop-1.0.0-jar-with-dependencies.jar
        com.clojurebook.JavaClojureInterop
Number of :a keywords in sample histogram: 8
Complete sample histogram: {:a 8, :c 4, :d 5, :b 4, :k 1, :e 3, :f 1}

Frequences of chars in 'I left my heart in san fransisco':
{\space 6, \a 3, \c 1, \e 2, \f 2, \h 1, \i 3, \l 1, \m 1,
 \n 3, \o 1, \r 2, \s 3, \t 2, \y 1}
Frequences of 500 random ints [0,10):
{0 60, 1 61, 2 55, 3 46, 4 37, 5 45, 6 47, 7 52, 8 49, 9 48}
```

Внимание. Существует две потенциальные проблемы, о которых следует упомянуть.

Для загрузки программного кода Clojure (как это делалось с помощью `require` в примере 9.21) необходимо, чтобы исходные файлы или скомпилированные файлы классов загружаемых пространств имен находились в пути поиска classpath.

Ссылки на переменные, к которым предполагается обращаться, в общем случае достаточно получить один раз (они обычно сохраняются в статических ссылках). Кроме того, если вы не ожидаете, что их значения изменятся[1], тогда вполне мудро будет получить их значения один раз (с помощью `fn()` или `deref()`). Это позволит избежать (пусть и небольших, но ощутимых) накладных расходов на поиск переменных во время выполнения.

Разобравшись с примером выше, вы сможете делать почти все что угодно с функциями и данными Clojure из Java. Теперь нам осталось лишь выяснить, как из Java можно использовать типы и протоколы, объявленные в Clojure.

[1] О том, когда и как могут изменяться значения переменных, рассказывается в разделе «Динамическая область видимости», в главе 4.

Использование классов, созданных с помощью deftype и defrecord

Каждая из этих форм[1] генерирует класс, доступный из Java. Благодаря этому имеется возможность создавать и использовать экземпляры таких классов в Java, как если бы они изначально были определены в Java.

Рассмотрим следующее пространство имен:

Пример 9.22. Определение нескольких классов с помощью deftype и defrecord

```
(ns com.clojurebook.classes)

(deftype Range                                                    ❶
  [start end]
  Iterable
  (iterator [this]
    (.iterator (range start end))))

(defn string-range                                                ❷
  "Возвращает экземпляр Range, опираясь на значения start и end,
   представленные в виде строк в списке/векторе/массиве."
  [[start end]]
  (Range. (Long/parseLong start) (Long/parseLong end)))

(defrecord OrderSummary                                           ❸
  [order-number total])
```

После его загрузки Clojure сгенерирует два класса, `com.clojurebook.classes.Range` и `com.clojurebook.classes.OrderSummary`. Их можно использовать из Java, как если бы они были написаны на Java; имена полей и методов этих классов будут даже распознаваться механизмом автодополнения интегрированной среды разработки.

Пример 9.23. Использование классов, представленных в примере 9.22, из Java

```
package com.clojurebook;

import clojure.lang.IFn;
```

[1] Форма `gen-class` во всех подробностях рассматривалась в разделе «Определение именованных классов» выше; а формы `deftype` и `defrecord` были описаны в главе 6.

```
import clojure.lang.RT;
import clojure.lang.Symbol;
import com.clojurebook.classes.OrderSummary;
import com.clojurebook.classes.Range;

public class ClojureClassesInJava {
     private static IFn requireFn = RT.var("clojure.core", "require").fn();
     static {
          requireFn.invoke(Symbol.intern("com.clojurebook.classes"));
     }

     private static IFn stringRangeFn = RT.var("com.clojurebook.classes",
          "string-range").fn();

     public static void main(String[] args) {
          Range range = new Range(0, 5);                                    ❶
          System.out.print(range.start + "-" + range.end + ": ");           ❷
          for (Object i : range) System.out.print(i + " ");                 ❸
          System.out.println();

          for (Object i : (Range)stringRangeFn.invoke(args))                ❹
               System.out.print(i + " ");
          System.out.println();

          OrderSummary summary = new OrderSummary(12345, "$19.45");         ❺
          System.out.println(String.format(
               "order number: %s; order total: %s",
               summary.order_number, summary.total));
          System.out.println(summary.keySet());
          System.out.println(summary.values());
     }
}
```

❶ Здесь создается экземпляр класса `Range`, объявленного с помощью `deftype`, и конструктору передаются два аргумента, как он того требует.

❷ Как описывается в разделе «Типы», в главе 6, классы, объявленные с помощью `deftype`, не реализуют какие-либо интерфейсы автоматически; однако мы можем обратиться к двум финальным (`final`) полям, используя имена, указанные в определении `deftype`, и...

❸ ...поскольку класс `Range` определен, как реализующий интерфейс `Iterable`, для него можно получить итератор и использовать этот итератор в цикле `for`, как любой другой экземпляр, реализующий интерфейс `Iterable`.

❹ Конструкторы классов, объявленных с помощью `deftype` и `defrecord`, часто могут требовать передачи значительного количества параметров определенных типов, в зависимости от того, как эти классы используются; то есть, обычно бывает предпочтительнее реализовать фабричную функцию, упрощающую создание экземпляров. Здесь мы используем фабричную функцию `string-range`, которая принимает любую деструктурируемую коллекцию с двумя строками и возвращает экземпляр класса `Range`, основанный на целочисленных значениях, полученных из этих строк. Это позволяет избежать необходимости вручную выбирать и анализировать данные, полученные из командной строки.

❺ Классы, объявленные с помощью `defrecord`, практически ничем не отличаются от классов, объявленных с помощью `deftype`, за исключением того, что для них предоставляется автоматическая реализация некоторых интерфейсов; здесь мы создаем экземпляр класса, объявленного с помощью `defrecord`, и проверяем возможность доступа к финальным (`final`) полям, а также реализацию по умолчанию пары методов интерфейса `java.util.Map`.

После компиляции нашего Java-класса, его можно запустить и посмотреть на полученные результаты:

```
% java -cp target/java-clojure-interop-1.0.0-jar-with-dependencies.jar
      com.clojurebook.ClojureClassesInJava 5 10
0-5: 0 1 2 3 4
5 6 7 8 9
order number: 12345; order total: $19.45
#{:order-number :total}
(12345 "$19.45")
```

Внимание. Когда необходима предварительная компиляция? Когда результаты работы любых форм Clojure (включая `deftype`, `defrecord`, `defprotocol` и `gen-class`), создающих классы, используются из Java, пространства имен, содержащие эти формы, должны предварительно компилироваться. Компилятор Java должен иметь готовые файлы классов, чтобы собрать код на Java, использующий Clojure-классы. Такой подход полностью отличается от использования формы `defrecord`, и других, в приложениях, написанных исключительно на языке Clojure – в таких приложениях Clojure просто генерирует и загружает необходимые классы в JVM прямо во время выполнения, минуя создание файлов на диске.
Предварительная компиляция обсуждалась в разделе «Предварительная компиляция», в главе 8, а проблемы, связанные с компиляцией гибридных проектов, рассматривались в разделе «Сборка гибридных проектов», в той же главе.

Реализация интерфейсов протоколов

Протоколы позволяют быстро создавать на языке Clojure очень гибкие модели предметной области[1]. Помимо возможности наследовать протоколы внутри программного кода Clojure для работы с существующими классами и интерфейсами Java, имеется также возможность наследовать протоколы Clojure в Java-классах без каких-либо изменений в коде на Clojure. Для поддержки этой возможности протоколы генерируют интерфейсы, которые можно реализовать в Java-классах. Например, ниже представлено пространство имен на Clojure, содержащее единственный протокол и две его реализации, одна для строк, а другая играет роль реализации по умолчанию, для работы со всеми объектами `Object`:

```
(ns com.clojurebook.protocol)

(defprotocol Talkable
  (speak [this]))

(extend-protocol Talkable
  String
  (speak [s] s)
  Object
  (speak [this]
    (str (-> this class .getName) "s can't talk!")))
```

Протокол `Talkable` определяет одну функцию `speak` и генерирует интерфейс `com.clojurebook.protocol.Talkable`, определяющий единственный метод `speak`. Этот интерфейс легко можно реализовать на Java:

Пример 9.24. Реализация Clojure-протокола на Java с импользованием сгенерированного интерфейса

```
package com.clojurebook;

import clojure.lang.IFn;
import clojure.lang.RT;
import clojure.lang.Symbol;
import com.clojurebook.protocol.Talkable;

public class BitterTalkingDog implements Talkable {
```

[1] Подробнее о протоколах рассказывается в главе 6.

```
    public Object speak() {                                              ❶
        return "You probably expect me to say 'woof!', don't you? Typical.";
    }

    Talkable mellow () {
        return new Talkable () {                                         ❷
            public Object speak() {
                return "It's a wonderful day, don't you think?";
            }
        };
    }

    public static void main(String[] args) {
        RT.var("clojure.core", "require").invoke(                        ❸
            Symbol.intern("com.clojurebook.protocol"));
        IFn speakFn = RT.var("com.clojurebook.protocol", "speak").fn();

        BitterTalkingDog dog = new BitterTalkingDog();

        System.out.println(speakFn.invoke(5));
        System.out.println(speakFn.invoke(
        "A man may die, nations may rise and fall, but an idea lives on."));
        System.out.println(dog.speak());
        System.out.println(speakFn.invoke(dog.mellow()));
    }
}
```

❶ Реализация в нашем классе метода `speak`, принадлежащего интерфейсу `Talkable`.

❷ Интерфейс, сгенерированный протоколом, ничем не отличается от других интерфейсов Java; здесь мы создаем и возвращаем экземпляр анонимного вложенного класса, реализующего интерфейс протокола.

❸ Чтобы воспользоваться расширениями `String` и `Object` протокола, необходимо загрузить пространство имен `com.clojurebook.protocol` и отыскать ссылку на переменную `speak`, определяющую протокол.

Как видите, Clojure обеспечивает возможность двунаправленных взаимодействий. Мы продемонстрировали только возможность использования из Clojure абстракций языка Java, однако то же самое возможно и в обратном направлении – из Java можно использовать абстракции языка Clojure.

Ниже приводятся результаты выполнения метода `main`, при вызове этого класса из командной строки:

```
% java com.clojurebook.BitterTalkingDog
java.lang.Integers can't talk!
A man may die, nations may rise and fall, but an idea lives on.
You probably expect me to say 'woof!', don't you? Typical.
It's a wonderful day, don't you think?
```

Сотрудничество

Несмотря на то, что Clojure предоставляет массу весьма привлекательных возможностей, тем не менее, для языков JVM весьма типично использовать все преимущества платформы, лежащей в их основе, включая ее зрелость, эффективность и надежность. Это дает вам возможность использовать в своих интересах обширнейшую экосистему Java, с ее библиотеками, фреймворками и сообществом, и вносить свой посильный вклад в ее развитие.

Глава 10. REPL-ориентированное программирование

Качество используемых инструментов играет чрезвычайно важную роль и может оказаться решающим фактором в освоении языка программирования, не говоря уже об успешности его использования. Интерактивная оболочка REPL в Clojure, которой мы коснулись в главе 1, является одним из важнейших инструментов и, как будет показано далее, одним из самых мощных.

Как подчеркивалось в самом начале, Clojure является компилирующим, а не интерпретирующим языком. Кроме того, как мы узнали в главе 5, компилятор Clojure остается доступным и во время выполнения, обеспечивая доступность всех возможностей языка во время выполнения и, соответственно, в оболочке REPL. Это означает, что:

- код, загружаемый и выполняемый в оболочке REPL (например, в среде разработки) будет работать точно так же и выполнять те же операции, что и код, загруженный из файлов на диске (например, в эксплуатационном окружении);
- оболочку REPL можно использовать для определения и переопределения любых конструкций в любой момент времени.

Благодаря этим особенностям, REPL является обязательной частью набора инструментов любого программиста на Clojure, чего нельзя сказать об аналогичных оболочках REPL и интерпретаторах для других языков. В этой главе мы исследуем некоторые возможности, поддерживаемые оболочкой REPL, которые могут коренным образом изменить ваши подходы к разработке программного обеспечения.

Интерактивная разработка

Интерактивная разработка – весьма широкий термин и может иметь самые разные интерпретации, так как большинство современных языков программирования предлагают тот или иной уровень интерактивности. Даже разработчики на Java имеют возможность

вычислять выражения в интерактивном режиме, например, когда приложение приостанавливается на контрольной точке в отладчике. И, конечно же, Ruby, Python и другие языки предоставляют свои оболочки REPL с разной степенью совершенства, хотя обычно они не интегрируются с другими инструментами (такими как текстовый редактор), могут запускаться только из командной строки, и их возможности часто ограничены в возможностях изменения или переопределения программного кода во время выполнения.

Интерактивная разработка на языке Clojure, поддерживаемая его оболочкой REPL, напротив, ослабляет все эти ограничения. По мере использования Clojure вы обнаружите, что наиболее продуктивным подходом оказывается создание приложений в интерактивном режиме с помощью постоянно открытого сеанса оболочки REPL, которая является весьма тонкой прослойкой между кончиками ваших пальцев и внутренними механизмами Clojure и JVM.

Пример 10.1. Небольшое «приложение» на основе библиотеки Swing

```
(ns com.clojurebook.fn-browser
  (:import (javax.swing JList JFrame JScrollPane JButton)
           java.util.Vector))

(defonce fn-names (->> (ns-publics 'clojure.core)                    ❶
                    (map key)
                    sort
                    Vector.
                    JList.))

(defn show-info [] )                                                 ❷

(defonce window (doto (JFrame. "\"Interactive Development!\"")
                  (.setSize (java.awt.Dimension. 400 300))
                  (.add (JScrollPane. fn-names))                     ❸
                  (.add java.awt.BorderLayout/SOUTH
                    (doto (JButton. "Show Info")                     ❹
                      (.addActionListener (reify
                                            java.awt.event.ActionListener
                                           (actionPerformed [_ e]
                                             (show-info))))))
                  (.setVisible true)))
```

❶ `fn-names` — это `JList`; модель, содержащая символы, представляющие имена всех общедоступных переменных в пространстве имен `clojure.core` и отсортированных в лексикографическом порядке.

❷ Функцию `show-info` пока оставим пустой. Мы дополним ее реализацию ниже.
❸ Здесь выполняется добавление в окно компонента списка внутри прокручиваемого контейнера.
❹ Здесь в окно добавляется кнопка с обработчиком события щелчка, который вызывает пока пустую функцию `show-info`.

Этот код можно загрузить в оболочку REPL или сохранить в файл *com/clojurebook/fn_browser.clj* где-нибудь в пути поиска classpath[1] и загрузить командой `(require 'com.clojurebook.fnbrowser)`. В любом случае, если предположить, что вы не используете Clojure на каком-нибудь сервере, не имеющем графической среды, вы увидите окно Swing как показано на рис. 10.1.

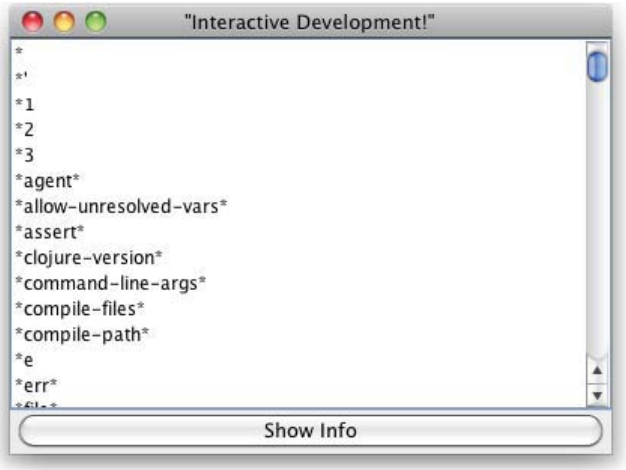

Рис. 10.1. Окно приложения из примера 10.1

Если только вы все еще не используете язык, основанный на дискретных этапах *создание кода* → *компиляция* → *отладка*, возможность запустить графический интерфейс из оболочки REPL не станет для вас большим открытием. Гораздо интереснее, что мы легко можем изменять выполняющуюся программу, просто загружая дополнительный код тем или иным способом.

---

[1] Об организации программного кода на языке Clojure и о понятии classpath рассказывается в главе 8.

Чтобы убедиться в этой возможности, заставим кнопку **Show Info** (Показать информацию) делать что-либо полезное. Для этого достаточно просто переопределить функцию `show-info`, вызываемую обработчиком события щелчка на кнопке, и реализовать в ней какие-либо действия, например, вывод документации к функции из `clojure.core`, выбранной в списке:

```
(in-ns 'com.clojurebook.fn-browser)

(import '(javax.swing JOptionPane JTextArea))

(defn show-info
  []
  (when-let [selected-fn (.getSelectedValue fn-names)]
    (JOptionPane/showMessageDialog
      window
      (-> (ns-resolve 'clojure.core selected-fn)          ❶
        meta                                              ❷
        :doc                                              ❸
        (JTextArea. 10 40)                                ❹
        JScrollPane.)
      (str "Doc string for clojure.core/" selected-fn)
      JOptionPane/INFORMATION_MESSAGE)))
```

❶ Отыскать функцию, выбранную в компоненте списка и находящуюся в пространстве имен `clojure.core`, можно с помощью функции `ns-resolve`. Она возвращает переменную...

❷ ...из которой извлекаются метаданные...

❸ ...и из этих метаданных – значение слота `:doc`, где Clojure сохраняет строку документации, указанную в определении переменной. Подробнее о строках документации рассказывается в разделе «Строки документации», в главе 4.

❹ Текст документации для функции затем используется как начальное содержимое компонента `JTextArea`, используемого для вывода «сообщения» в диалоге `JOptionPane`.

Этот код можно загрузить в оболочку REPL или сохранить его в файле *fn_browser.clj* и перезагрузить его командой `(require 'com.clojurebook.fn-browser :reload)`[1]. В любом случае, после переопределения функ-

[1] В последнем случае импорт `JOptionPane` и `JTextArea` можно было бы добавить в объявление `:import` в форме `ns` в файле; эти дополнительные зависимости будут импортированы в пространство имен при загрузке файла. Если указать флаг `:reload-all`, это приведет к транзитивной перезагрузке всех зависимостей в объявлениях `:require` и `:use`.

ции `show-info`, обработчик события щелчка на кнопке будет вызывать вновь определенную функцию *и при этом не потребуется пересоздавать, изменять или как-то еще касаться реализации самой кнопки.*

Выберите в списке функцию, имеющую документацию, щелкните на кнопке **Show Info** (Показать информацию) и на экране появится диалог с этой документацией, как показано на рис. 10.2.

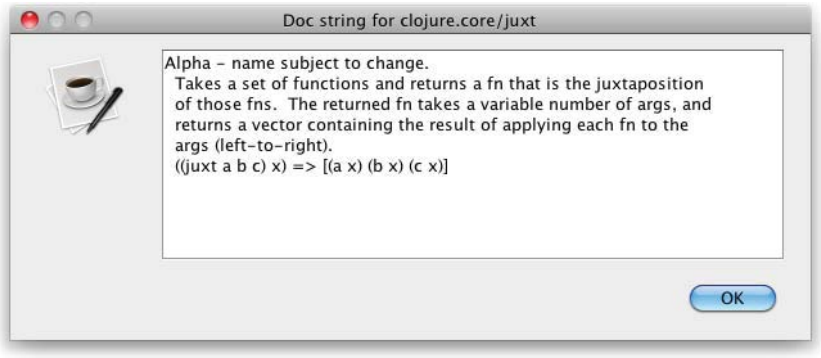

Рис. 10.2. Диалог с описанием выбранной функции

Использование defonce, чтобы исключить затирание переменных

В файле с некоторым кодом, который вы можете загружать и перезагружать в каким-либо способом – в оболочке REPL или через параметры `:reload` и `:reload-all` в формах `use` и `require` – у вас могут иметься определения переменных, значения которых было бы *нежелательно* вычислять повторно. Например, было бы нежелательно переопределять переменные `window` и `fn-names` в примере 10.1, так как это будет приводить к созданию нового окна и нового компонента списка при каждой перезагрузке файла.

Решить эту проблему можно с помощью формы `defonce`. Подобно `def` и родственным ей формам, `defonce` определяет значение переменной в текущем пространстве имен, но не переопределяет переменную, если она уже имеет некоторое значение. Благодаря этому `defonce` позволяет смешивать определения переменных, которые должны хранить фиксированное значение на протяжении всего времени жизни приложения (как `window` в нашем примере или, к примеру, пул соединений с базой данных в веб-приложении), с определениями других переменных, которые могут переопределяться на протяжении всего цикла разработки, и даже на этапе эксплуатации.

Это был очень простой пример, но мы надеемся, что он смог продемонстрировать гибкость, которую дает использование интерактивной оболочки REPL. Интерактивную оболочку также удобно использовать для доработки пользовательских интерфейсов без перезапуска или перестроения окна. Той же гибкостью и немедленной обратной связью можно пользоваться при разработке алгоритмов, выполняющих массивные вычисления, или сложных моделей предметной области.

Постоянное изменяющееся окружение

После демонстрации выше может сложиться впечатление, что это всего лишь небольшое улучшение, дополняющее типичный процесс разработки, особенно у тех, кто имеет опыт работы с такими языками, как Ruby, Python и PHP. В конце концов, все эти и многие другие языки позволяют загружать код в интерактивном режиме и переопределять различные конструкции в ходе продолжительного сеанса работы. Однако в действительности улучшения уходят гораздо глубже:

В Clojure процесс разработки не требует создания файлов. В Python, Ruby и PHP для загрузки программного кода необходимо, чтобы он хранился на диске[1]. Это вполне разумный подход и при использовании Clojure – как указывалось в описании к предыдущему примеру, для повторной загрузки прежде загруженного пространства имен можно использовать параметр `:reload` – но он не является обязательным. Загрузить код в процесс Clojure можно вообще не касаясь диска, например, непосредственно вводя его в оболочку REPL (с клавиатуры) или с помощью специализированных команд среды разработки на Clojure загружать файлы, пространства имен или целые проекты.

Динамизм языка Clojure явно поддерживается языковыми конструкциями и средой выполнения. Многие основные аспекты архитектуры Clojure явно способствуют (или *не препятствуют*) динамическому переопределению его конструкций во время выполнения. Воплощение пространств имен и переменных, отсутствие требований, обязывающих хранить код на диске или в буфере текстового редактора, широкое использование байт-кода, сгенерированного во

---

[1] В случае с Ruby это не совсем верно, но многие инструменты и правила, окружающие Ruby, подталкивают программиста к этому.

время выполнения, и скомпилированных классов, сокращение дистанции между этапом компиляции и этапом выполнения – эти и многие другие метаособенности языка Clojure сообща делают возможным использовать его как постоянный холст, на котором вы можете изобразить картину своих представлений.

Это может казаться пустой рекламой... пока вы не попробуете использовать среду разработки на Clojure, снабженную хотя бы минимальными возможностями REPL. Такие среды позволяют связать текстовый редактор с одним или более запущенными сеансами REPL. Эта связь дает возможность быстро переходить от правки программного кода в редакторе – откуда нажатием одной клавиши можно отправить единственное выражение или целый файл в постоянно открытый сеанс REPL – к взаимодействию со средой выполнения Clojure, ассоциированной с этой оболочкой REPL, чтобы немедленно проверить результаты, поэкспериментировать с не до конца сформировавшимися идеями, не «пачкая» исходный код проекта, и вообще исследовать среду выполнения Clojure, чтобы проверить свою работу и получить представление о том, куда двигаться дальше. Для программистов на Clojure является обычным делом использование одного и того же процесса JVM/Clojure в течение нескольких дней, где они последовательно изменяют состояние среды выполнения и приложений, пока не добьются желаемых результатов.

Такой способ разработки, который иногда называют *потоком* (flow), помогает сконцентрировать внимание на задаче и получить более ясное представление о путях ее решения, а доступность важной информации расширяет возможности и усиливает ощущение контроля. Разумеется, Clojure не единственный язык, позволяющий использовать такой подход – описываемые преимущества доступны программистам на самых разных языках. Однако с полной уверенностью можно сказать, что Clojure открывает более широкие возможности, чем большинство других языков, в немалой степени благодаря богатству возможностей его интерактивной оболочки REPL, обеспечивающих более тесную связь между программами и программистами. Этот подход подробно обсуждался на конференции 2010 Clojure Conj[1]:

---

[1] Видеозапись этого обсуждения и слайды можно найти по адресу: http://blip.tv/clojure/tom-faulhaber-lisp-functional-programming-and-the-state-of-flow-4539472.

> Программирование с применением REPL в некоторых отношениях напоминает сотрудничество с наставником. Благодаря богатству возможностей REPL мы имеем возможность исследовать текущее состояние, наблюдать, что происходит с машиной и как действуют наши алгоритмы.
>
> – Том Фаульхабер (Tom Faulhaber),
> «Lisp, Functional Programming, and the State of Flow»

Как только вы освоите оболочку REPL и получите представление о том, как можно использовать ее в повседневной практике программирования, наступит время подкрепить этот опыт, взяв на вооружение соответствующие инструменты.

Инструменты

Из-за сложности Java, под «инструментами для Java» всегда подразумевались по-настоящему интегрированные среды разработки (Integrated Development Environment, IDE), такие как Eclipse и IntelliJ IDEA, обладающие интеллектуальными средствами автодополнения кода, рефакторинга, визуализации иерархии классов и другими особенностями, которые представляют собой насущную необходимость для большинства программистов на Java. Напротив, для использования динамических языков программирования (включая Python и Ruby) обычно требуется только текстовый редактор и командная строка. Большинство программистов на Clojure придерживаются последней модели[1], с одной существенной разницей: наличие текстового редактора, обладающего возможностью интеграции с Clojure REPL, – всегда предпочтительнее иметь редактор, хорошо интегрирующийся с другими инструментами, – играет важную роль, но не основную.

К счастью поддержка Clojure доступна в различных популярных редакторах (таких как Emacs, vim, TextMate, jEdit и так далее) и IDE (таких как Eclipse, Intellij IDEA и NetBeans). Все эти инструменты достаточно просты в освоении[2]; но при прочих равных условиях мы рекомендуем использовать инструменты, наиболее удобные для

---

[1] Многие среды программирования для Clojure поддерживают такие возможности, как автодополнение кода, но здесь мы говорим о минимальных требованиях, а не о том, что существует или желательно.

[2] Дополнительные указания можно найти по адресу: http://dev.clojure.org/display/doc/Clojure+Tools.

вас и лучше всего согласующиеся с вашим стилем работы[1]. Чтобы дать вам отправную точку для сравнения, ниже мы познакомимся с двумя наиболее популярными инструментами, используемыми в сообществе, каждый из которых представляет неповторимый подход к поддержке Clojure: Eclipse и Emacs.

Для начала познакомимся с наиболее простыми и практичными инструментами, предоставляемыми всеми оболочками REPL, знать которые совершенно необходимо, чтобы расширить свой опыт программирования.

Оболочка REPL

Оболочка Clojure REPL – это инструмент повседневного пользования, а это означает, что ее возможностями вы будете пользоваться каждый день. Неважно, используете ли вы простенький текстовый редактор и оболочку REPL, запущенную в отдельном окне терминала, или большую интегрированную среду разработки со встроенной оболочкой REPL, описываемые ниже возможности будут доступны всегда[2].

Переменные, создаваемые оболочкой REPL. Существует несколько переменных, которые обычно создаются только внутри сеанса REPL и добавляют дополнительные удобства, необходимые в интерактивной среде.

---

[1] Следует отметить, что если вам удобно пользоваться *Блокнотом* (*notepad.exe*), используйте этот редактор, главное чтобы у вас под рукой имелась открытая оболочка REPL в соседнем окне терминала. На наш взгляд существуют более удачные варианты, и вы можете подумать о них позднее, но нет ничего хуже, чем одновременно изучать новый язык программирования и осваивать новые инструменты.

[2] Если вы собираетесь запускать оболочку REPL из командной строки (то есть, с помощью команды `java`), не прибегая к использованию REPL, входящей в состав Leiningen, Counterclockwise, Emacs или любого другого инструмента для Clojure, Тогда вам желательно будет получить библиотеку JLine (http://jline.sourceforge.net) или `rlwrap` (http://utopia.knoware.nl/~hlub/rlwrap/). Оболочка REPL, встроенная в Clojure, не поддерживает такие особенности, как повторный вызов команды (например, она не поддерживает возможность нажать клавишу со стрелкой вверх, чтобы вернуться к предыдущей команде, выполненной в REPL) или редактирование строки команды, еще не отправленной на выполнение; обе библиотеки, JLine и `rlwrap`, добавляют поддержку этих возможностей во встроенную оболочку REPL.

- `*1`, `*2` и `*3` хранят значения самых последних вычисленных выражений, например, `*1` соответствует переменной _ в Ruby и Python;
- `*e` представляет последнее необработанное исключение, возникшее в сеансе REPL; она напоминает кортеж `sys.last_type`, `sys.last_value` и `sys.last_traceback` в Python.

Эти переменные, управление которыми осуществляется автоматически, могут оказаться особенно удобными при исследовании прикладных интерфейсов и данных:

```
(split-with keyword? [:a :b :c 1 2 3])
;= [(:a :b :c) (1 2 3)]
(zipmap (first *1) (second *1))
;= {:c 3, :b 2, :a 1}
(apply zipmap (split-with keyword? [:a :b :c 1 2 3]))
;= {:c 3, :b 2, :a 1}
```

`clojure.repl/pst` выведет трассировку стека для любых возникших исключений, но по умолчанию ограничивается только исключением, хранящимся в `*e`:

```
(throw (Exception. "foo"))
;= Exception foo user/eval1 (NO_SOURCE_FILE:1)
(pst)
; Exception foo
;    user/eval1 (NO_SOURCE_FILE:1)
;    clojure.lang.Compiler.eval (Compiler.java:6465)
; ...
```

clojure.repl. Пространство имен `clojure.repl` содержит множество утилит, очень удобных для использования в REPL. Выше вы видели функцию `pst`; кроме нее существует еще функция `apropos`, которая выводит список функций в загруженном пространстве имен, имена которых совпадают с указанным регулярным выражением или строкой:

```
(apropos #"^ref")
;= (ref-max-history refer-clojure ref-set
;=  ref-history-count ref ref-min-history refer)
```

`find-doc` делает почти то же самое, только поиск выполняется внутри документации и она выводит всю информацию, связанную с переменной, где найдено совпадение.

Также имеется функция `source`, которая выводит код функции, загруженной из исходных текстов:

```
(source merge)
; (defn merge
;    "Returns a map that consists of the rest of the maps conj-ed onto
;    the first. If a key occurs in more than one map, the mapping from
;    the latter (left-to-right) will be the mapping in the result."
; {:added "1.0"
;  :static true}
; [& maps]
; (when (some identity maps)
;    (reduce1 #(conj (or %1 {}) %2) maps)))
```

Наконец, имеется функция `doc`, которая выводит документацию только для указанной переменной; и функция `dir`, которая выводит список общедоступных переменных, объявленных в указанном пространстве имен:

```
(require 'clojure.string)
;= nil
(dir clojure.string)
; blank?
; capitalize
; escape
; join
; lower-case
; replace
; replace-first
; reverse
; split
; split-lines
; trim
; trim-newline
; triml
; trimr
; upper-case
```

Крайне редко можно встретить Clojure REPL, которая не загружает пространство имен `clojure.repl` с его удобными функциями.

Интроспекция пространств имен

Пространства имен – это самостоятельные программные компоненты, такие же, как любые другие структуры данных. Существует

ряд функций, позволяющих исследовать и изменять пространства имен в оболочке REPL; рассмотрим некоторые из них[1].

Обратите внимание, что большую часть времени вам вообще не придется прикасаться к этим функциям. Однако, если вы по ошибке определили в пространстве имен какие-либо функции или данные, с помощью этих функций вы сможете быстро отыскать и удалить проблемные определения. Это может помочь вам выйти из некоторых ситуаций, когда иначе потребовалось бы перезапустить приложение или сеанс REPL, таких как необходимость определить тип с помощью `deftype`, имя которого совпадает с именем существующего Java-класса и уже импортированного в пространство имен.

ns-map, ns-imports, ns-refers, ns-publics, ns-aliases, ns-interns. Все эти функции возвращают ассоциативный массив с символами, связанными в указанном пространстве имен либо с переменными, либо с импортированными классами. То есть, там, где для регистрации символов в пространстве имен использовались `refer`, `import` и `def`, эти функции сообщат об имеющихся связях.

```
(ns clean-namespace)
;= nil
(ns-aliases *ns*)
;= {}
(require '[clojure.set :as set])
;= nil
(ns-aliases *ns*)
;= {set #<Namespace clojure.set>}
(ns-publics *ns*)
;= {}
(def x 0)
;= #'clean-namespace/x
(ns-publics *ns*)
;= {x #'clean-namespace/x}
```

ns-unmap, ns-unalias. Первую функцию можно использовать для удаления связей между символами и переменными или импортированными классами, а вторую – для удаления псевдонима пространства именwhile.

---

[1] Если вы пожелаете самостоятельно заняться исследованиями, команда `(apropos #»(ns-|-ns)»)` позволит получить более полный список таких функций.

```
(ns-unalias *ns* 'set)
;= nil
(ns-aliases *ns*)
;= {}
(ns-unmap *ns* 'x)
;= nil
(ns-publics *ns*)
;= {}
```

remove-ns. Это – «ядерная бомба» в управлении пространствами имен. В отличие от макроса `ns`[1], создающего пространство имен, функция `remove-ns` удаляет указанное ей пространство имен.

```
(in-ns 'user)
;= #<Namespace user>
(filter #(= 'clean-namespace (ns-name %)) (all-ns))
;= (#<Namespace clean-namespace>)
(remove-ns 'clean-namespace)
;= #<Namespace clean-namespace>
(filter #(= 'clean-namespace (ns-name %)) (all-ns))
;= ()
```

То есть, весь код и данные, имевшиеся в пространстве имен, станут недоступны после его удаления и будут *утилизированы* сборщиком мусора Конечно, если какие-то ссылки на функции, протоколы или данные, объявленные в удаленном пространстве имен, будут храниться где-то в другом месте, они останутся доступны.

> **Примечание. Структурное редактирование исходного кода на Clojure.** Мы являемся решительными поборниками равноправия, когда речь заходит о выборе текстового редактора, однако есть один настолько превосходный инструмент редактирования исходного программного кода на языке Clojure, что мы не можем не упомянуть о нем: paredit – модуль редактирования, родившийся в недрах сообщества пользователей Emacs, упрощающий редактирование s-выражений и являющийся одним из инструментов, которые многие программисты на Clojure считают совершенно необходимыми.
>
> Большинство высококачественных редакторов с поддержкой Java предоставляют различные дополнительные возможности, такие как автоматическая вставка парных скобок или расширение вашего выбора до включения вмещающего элемента, выражения или области видимости. Модуль paredit поддерживает эквивалентные возможности для языка Clojure,

[1] Или намного реже используемого макроса `create-ns`.

а большинство его реализаций не ограничиваются понятиями, поддерживаемыми в редакторах для Java, и включают такие возможности, как перемещение курсора или области выделения через целые s-выражения за раз, перемещение целых выражений и автоматическое заключение выбранных выражений в квадратные, фигурные или круглые скобки, позволяя обеспечить структурную целостность кода.

Проще говоря, если вы когда-либо сталкивались со сложностями при редактировании программного кода на языке Clojure – например, испытывали затруднения с выделением отдельных s-выражений или с соблюдением парности скобок – вам определенно стоит поискать реализацию модуля paredit для вашего редактора или перейти на использование другого редактора, предоставляющего их.

Eclipse

Eclipse – в паре с Counterclockwise[1], модулем расширения, обеспечивающем поддержку Clojure в Eclipse – предоставляет исчерпывающий набор возможностей, необходимых для разработки программ на языке Clojure: редактирование, автодополнение, интеграция с оболочкой REPL, интроспекция, отладка и профилирование. Eclipse – не самая легковесная среда разработки, но взамен она дает полноценный набор инструментов, более удобных в использовании для многих программистов, которые ценят знакомые и узнаваемые пользовательские интерфейсы. Кроме того, если в рамках одного проекта вам потребуется писать код не только на языке Clojure, но и на языке Java, вам сложно будет отказаться от уровня поддержки Java, предлагаемой только такими интегрированными средами разработки, как Eclipse.

Редактирование кода на Clojure. Поддержка редактирования программного кода на Clojure в Counterclockwise не такая обширная, как в Emacs, тем не менее, она ничем не уступает лучшим образцам. Этот модуль расширения частично реализует возможности paredit[2], подсветку синтаксиса и полный набор возможностей редактирования текста, унаследованных от Eclipse. Каждый редактор с поддержкой Clojure также поддерживает и стандартное представление **Outline** (Схема документа) в Eclipse, предоставляя для него список

---

[1] Для начала обратитесь по адресу: http://dev.clojure.org/display/doc/Getting+Started+with+Eclipse+and+Counterclockwise; адрес домашней страницы проекта Counterclockwise: http://code.google.com/p/counterclockwise/.

[2] Описывается выше, в примечании «Структурное редактирование исходного кода на Clojure».

всех выражений верхнего уровня (обычно определения функций) в текущем файле:

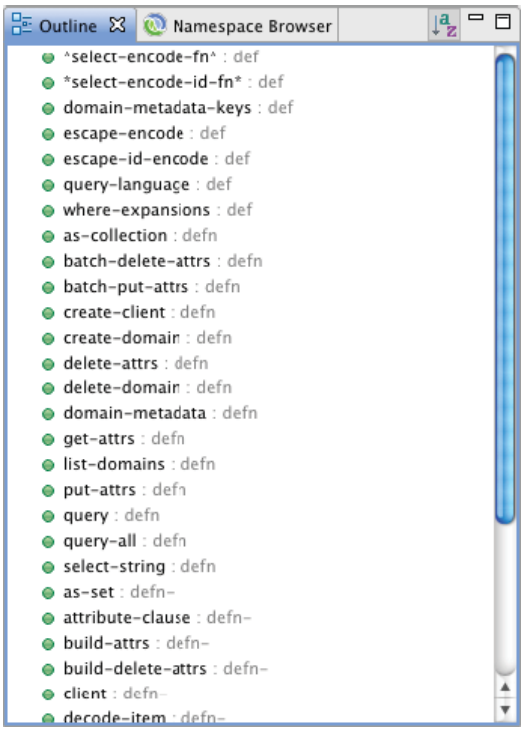

Рис. 10.3. Вид окна представления **Outline** (Схема документа)

Eclipse и Counterclockwise поддерживают вывод множества подсказок, чтобы помочь вам узнать, какие команды доступны и какие горячие комбинации клавиш могут использоваться для выполнения операций в разных контекстах (которые могут отличаться в разных операционных системах). Чтобы получить справку, в любой момент можно обратиться к меню **Clojure** (или к контекстному меню в редакторах), которое Counterclockwise добавляет в Eclipse, а также к справочной странице редактора Clojure, открывающейся при выборе пункта меню **Help → Dynamic Help** (Справка → Динамическая справка), когда открыт какой-нибудь файл с исходным кодом на языке Clojure.

Интеграция с REPL. Для интеграции со своей реализацией REPL в Counterclockwise используется nREPL[1] (библиотека, реализующая серверную и клиентскую части а Clojure REPL для использования другими инструментами и легко встраиваемая в приложения на языке Clojure). Она позволяет подключаться и взаимодействовать с любыми приложениями на Clojure со встроенным сервером nREPL, включая все процессы Clojure, запускаемые с помощью Eclipse и Counterclockwise. Помимо возможности вычислять выражения, оболочка REPL в Counterclockwise интегрируется с редактором, давая возможность загружать код из открытых файлов и выделенных фрагментов, а также поддерживает историю команд и автодополнение кода, как показано на рис. 10.4.

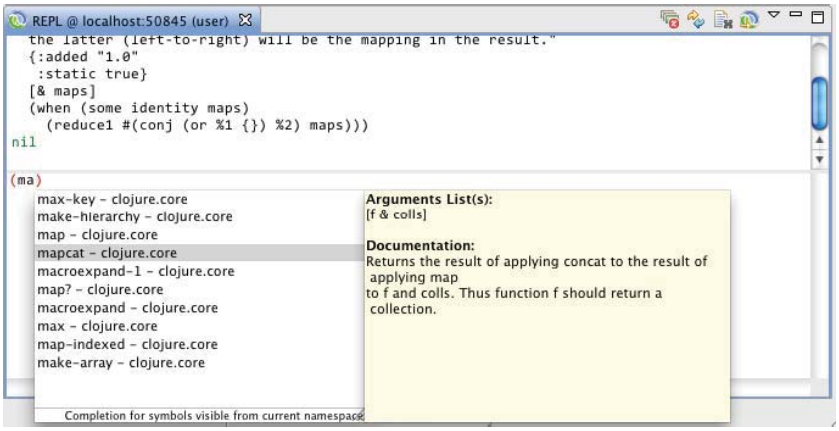

Рис. 10.4. Интегрированная оболочка REPL в Counterclockwise

После запуска оболочки REPL, та же самая поддержка автодополнения кода и возможность перехода к определению становятся доступны в редакторе, связанном с открытым сеансом REPL.

Обзор пространств имен. Counterclockwise также поддерживает графический браузер пространств имен. Это представление позволяет просматривать и выполнять поиск по всем пространствам имен и переменным, загруженным и объявленным в окружении Clojure в текущей оболочке REPL.

---

[1] http://github.com/clojure/tools.nrepl.

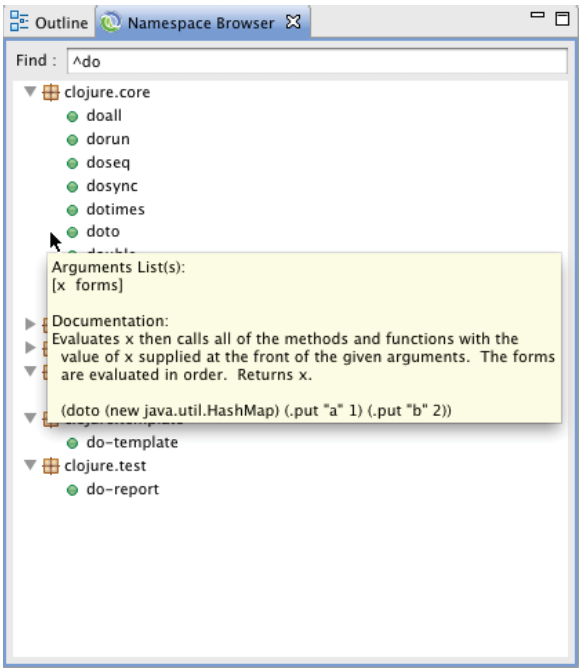

Рис. 10.5. Браузер пространств имен в Counterclockwise

Список переменных можно фильтровать (используя регулярные выражения, как показано на рис. 10.5). При наведении указателя мыши на имя переменной, выводится всплывающая подсказка с текстом документации; двойной щелчок открывает соответствующий файл и выполняет переход к определению переменной.

Emacs

Emacs (http://www.gnu.org/s/emacs/) – это мощный и расширяемый текстовый редактор, ставший надежной опорой программирования на языке Lisp за прошедшие десятилетия. Поддержка Clojure в Emacs обеспечивается комбинацией ряда инструментальных модулей и библиотек.

Существует два основных режима работы с программным кодом на языке Clojure в Emacs: `inferior-lisp` и SLIME. Оба зависят от наличия установленного модуля `clojure-mode`.

Термины Emacs

Emacs создавался задолго до появления большинства современного программного обеспечения и это отразилось на терминологии, используемой для обозначения его команд и функций.

Буфер
Именованный объект Emacs, представляющий нечто, доступное для редактирования, обычно файл в файловой системе, но это может быть также оболочка REPL или отладчик, например.

Окно
Панель или область редактирования в Emacs. В Emacs можно просматривать сразу несколько файлов (или несколько файлов + буфер REPL).

M-x foo
Нажав и удерживая клавишу **Meta** (обычно **Alt** или **Option**) на клавиатуре, нажать и отпустить клавишу **x**, затем отпустить клавишу **Meta**. После этого появится приглашение к вводу, где можно ввести команду `foo` и нажать клавишу **Enter**.

C-k
Нажав и удерживая клавишу **Control** (обычно **Control** или **Ctrl**) на клавиатуре, нажать и отпустить клавишу **k**, затем отпустить клавишу **Control**.

C-M-x
Нажав и удерживая клавиши **Control** и **Meta**, нажать и отпустить клавишу **x**, затем отпустить клавиши **Control** и **Meta**.

C-x C-e
Нажав и удерживая клавишу **Control**, нажать и отпустить клавишу **x**, затем нажать и отпустить клавишу **e**, а затем отпустить клавишу **Control**.

Более подробную информацию можно найти в руководстве по редактору Emacs: http://www.gnu.org/software/emacs/manual/emacs.html[1].

clojure-mode и paredit

При использовании любого из режимов, `inferior-lisp` или SLIME, наличие `clojure-mode` и paredit приобретает особую важность.

Модуль `clojure-mode` (https://github.com/technomancy/clojure-mode) добавляет в Emacs поддержку особенностей, имеющих отношение

---

[1] На сайте IBM developerWorks можно найти серию статей «Среда редактирования Emacs» на русском языке. Первая статья в цикле: http://www.ibm.com/developerworks/ru/edu/au-emacs1/index.html. – *Прим. перв.*

к языку Clojure, таких как подсветка синтаксиса, оформление отступов и средства навигации по программному коду. Он также включает в себя дополнительный модуль `clojure-test-mode`, реализующий поддержку автоматического тестирования с помощью `clojure.test`. Инструкции по настройке `clojure-mode` можно найти на главной странице проекта.

Помимо `clojure-mode` имеется также вышеупомянутый модуль `paredit.el` (http://www.emacswiki.org/emacs/ParEdit), обеспечивающий поддержку автоматического добавления парных скобок; он уже включен в состав Emacs.

inferior-lisp

`inferior-lisp`[1] является наиболее часто используемым режимом работы с программным кодом на языке Clojure в Emacs. Он применяется для запуска оболочки Clojure REPL в дочернем процессе и отображает ее в буфере Emacs. Эту оболочку REPL можно использовать как и любую другую, запущенную из командной строки, но сверх того `inferior-lisp` позволяет в интерактивном режиме посылать программный код на Clojure из открытого файла в буферы REPL.

Одно из преимуществ `inferior-lisp` перед SLIME в том, что режим `inferior-lisp` встроен в Emacs, а его настройка заключается в установке единственной переменной в конфигурационном файле Emacs (эту переменную можно также определить, нажав комбинацию клавиш **M-:**), чтобы сообщить Emacs, как вызывать Clojure:

```
(setq inferior-lisp-program "lein repl")
```

Значением переменной `inferior-lisp-program` может быть строка с командой запуска оболочки REPL, например, с помощью Leiningen, как показано выше, Java или любого другого инструмента, такого как `mvn clojure:repl`. После настройки переменной, оболочку Clojure REPL можно запустить комбинацией клавиш **C-c C-z**. Обратите внимание, что команда, указанная в переменной `inferior-lisp-program`, запускается в текущем каталоге Emacs. В зависимости от способа запуска Emacs, вам может понадобиться изменить его текущий каталог (с помощью команды **M-x cd**) перед запуском процесса `inferior-lisp`. Это – самый простой и самый быстрый способ настроить и запустить Clojure в Emacs.

---

[1] Названный так, чтобы подчеркнуть тот факт, что Lisp используется в подпроцессе, запущенном редактором Emacs: http://www.gnu.org/s/libtool/manual/emacs/External-Lisp.html.

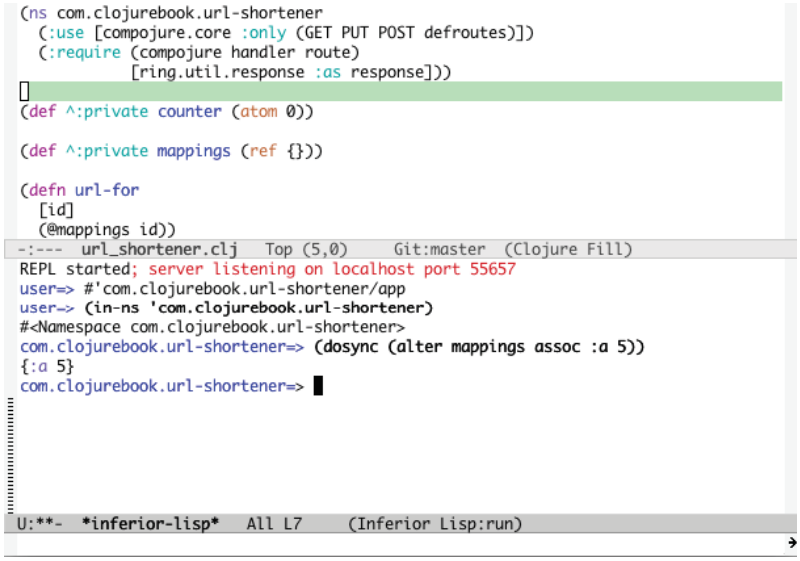

Рис. 10.6. Интерактивная оболочка REPL, запущенная в Emacs

Возможностей редактирования в этом режиме поддерживается значительно меньше, чем в режиме SLIME, но кому-то их будет вполне достаточно. Обычно при редактировании исходного кода на Clojure в Emacs открывается одно-два окна Emacs с исходными текстами и дополнительное окно Emacs с буфером REPL. Код можно редактировать в одном окне и затем отправлять в REPL посредством одной или нескольких команд. Эту возможность поддерживают оба режима, `inferior-lisp` и SLIME, но команды в них отличаются (см. табл. 10.1).

Таблица 10.1. inferior-lisp

Комбинация клавиш	Команда для M-x	Описание
C-c C-z	run-lisp	Запускает настроенный процесс `inferior-lisp` вызовом команды, указанной в переменной `inferior-lisp-program`
C-M-x	lisp-eval-defun	Вычисляет форму верхнего уровня (такую как `defn`) под курсором
C-x C-e	lisp-eval-last-sexp	Вычисляет s-выражение, предшествующее курсору
C-c C-l	clojure-load-file	Загружает в оболочку REPL текущий файл целиком

SLIME

SLIME[1] – это библиотека для Emacs, обеспечивающая поддержку расширенных возможностей редактирования и собственную реализацию REPL для различных диалектов Lisp, включая (но не ограничиваясь) Clojure. Режим SLIME обладает гораздо более широкими возможностями, в сравнении с режимом `inferior-lisp`, включая сохранение истории команд, автодополнение кода, интроспекция пространств имен, отладка, и многое другое.

> **Внимание.** Расширяемость Emacs может быть одновременно и благом и наказанием. Emacs, вероятно, способен делать то, что вы могли бы ожидать от текстового редактора, но настройка 'n[действий может оказаться непростой задачей.
> В частности, инструкции по установке SLIME и других инструментов, имеющих отношение к Clojure, в Emacs быстро и неоднократно изменялись, одновременно с развитием поддержки Clojure в Emacs. В Интернете сегодня можно найти массу устаревших статей в блогах и в Википедии, описывающих устаревшие методики настройки поддержки Clojure в Emacs. Мы предлагаем сразу начинать с документа http://dev.clojure.org/display/doc/Getting+Started+with+Emacs, где приводятся инструкции по установке и настройке. Это – Вики-страница, постоянно обновляемая разработчиками различных проектов, обеспечивающих поддержку Clojure в Emacs.

Чтобы иметь возможность пользоваться режимом SLIME, в Clojure-проекте должна быть настроена поддержка сервера *swank*; как и nREPL, swank – это протокол взаимодействия с реализацией REPL в SLIME. Проще всего эту настройку выполнить с помощью `lein` – достаточно добавить `swank-clojure` (реализация swank на языке Clojure), как зависимость этапа разработки в проект Leiningen:

```
lein plugin install lein-swank 1.4.3
```

... или добавить в файл *project.clj* вектор расширений:

```
[lein-swank "1.4.3"]
```

Режим SLIME можно вызвать в Emacs командой **M-x clojure-jack-in** после открытия в текущем буфере файла, для которого требуется запустить REPL. Если возникнут какие-то проблемы, обращайтесь

[1] Аббревиатура от Superior Lisp Interaction Mode for Emacs (режим Emacs для разработки приложений на Common Lisp): http://common-lisp.net/project/slime/.

по адресу: http://dev.clojure.org/display/doc/Getting+Started+with+Emacs, где всегда можно найти самые свежие инструкции.

В SLIME можно использовать все клавиатурные привязки, поддерживаемые режимом `inferior-lisp`, но первый предоставляет большее количество команд для передачи кода из буфера в оболочку REPL и выполнения различных операций внутри окружения Clojure/SLIME в контексте текущего файла. Некоторые из них перечислены в табл. 10.2.

Таблица 10.2. Часто используемые команды SLIME и их клавиатурные привязки

Комбинация клавиш	Команда для M-x	Описание
C-c C-c	slime-compile-defun	Вычисляет форму верхнего уровня под курсором
C-c C-k	slime-compile-and-load-file	Загружает текущий файл целиком
M-.	slime-edit-definition	Переходит к определению символа под курсором
C-c C-m	slime-macroexpand-1	Вызывает `macroexpand-1` для выражения, следующего за курсором
C-c M-m	slime-macroexpand-all	Вызывает `macroexpand-all` для выражения, следующего за курсором
C-c I	slime-inspect	Выводит пригодное для навигации представление значения указанного символа или класса

Инспектор объектов. Поиск за пределами границ структуры данных или API иногда может оказаться весьма сложной задачей, особенно при использовании некоторых библиотек Java. Режим SLIME поддерживает возможность визуализации коллекций Clojure, а также объектов и классов Java посредством *инспектора объектов*. Просто нажмите комбинацию клавиш **C-c I** и введите символ или выражение, значение которого желаете исследовать, или поместите текстовый курсор в пределы интересующего символа и нажмите комбинацию клавиш – перед вами откроется панель инспектора объектов с информацией о символе, как показано на рис. 10.7.

Ту же операцию поиска можно применять к значениям и классам, включенным в вывод инспектора объектов (или просто нажмите клавишу **Enter**, поместив предварительно курсор в пределы интересующего символа или значения). Это позволяет легко перемещаться по сложным API или большим структурам данных.

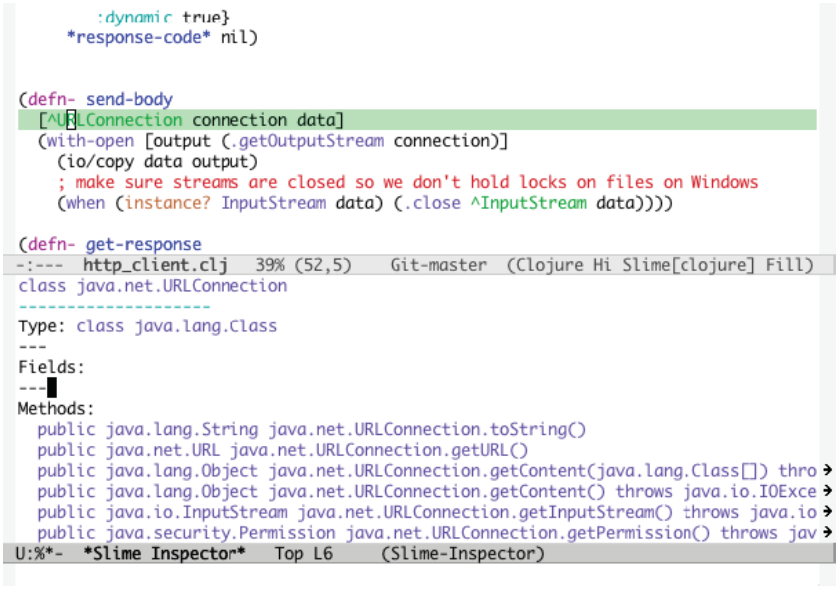

Рис. 10.7. Инспектор объектов

Отладка. Когда в SLIME REPL возникает исключение, выполняется переход в отладчик SLIME. Отладчик можно также вызвать вручную, установив контрольную точку в коде:

```
(defn debug-me
  [x y]
  (let [z (merge x y)]
    (swank.core/break)))
```

Выполнив выражение `(debug-me {:a 5} {"b" 5/6})`, вы увидите картину, как показано на рис. 10.8.

Щелчок мышью на какой-либо части трассировки приведет к выводу локальных привязок и их значений, как в инспекторе объектов, где так же можно производить исследование этих значений, таким же способом.

Конечно, когда активизируется отладчик, Clojure и оболочка REPL продолжают оставаться активными, что позволяет вычислять произвольные выражения: исследовать или изменять текущее состояние программы, определять и переопределять функции, и так далее. В частности, локальные привязки в области видимости

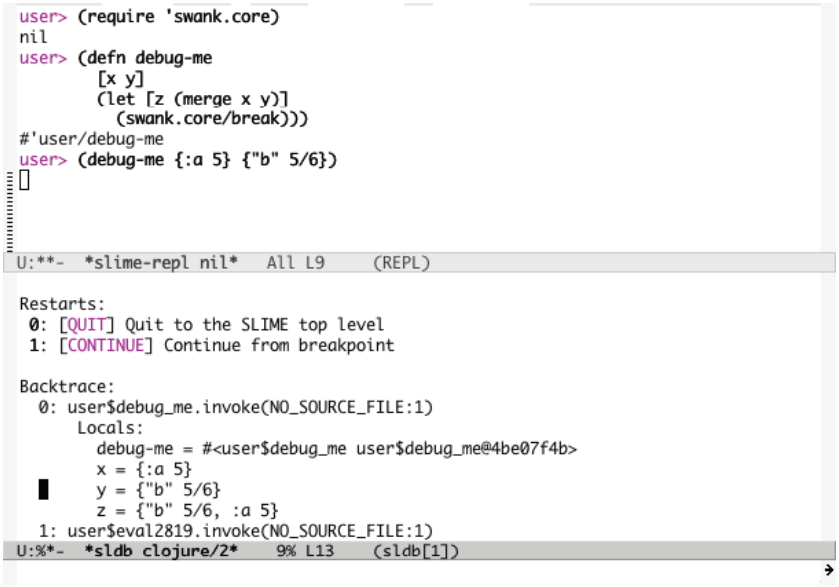

Рис. 10.8. Отладчик SLIME в действии

вызова `swank.core/break` доступны для исследования и изменения в SLIME REPL, даже при том, что поток выполнения, встретивший контрольную точку, приостановлен. Это огромное преимущество программирования в интерактивной среде.

Отладка, мониторинг и исправление программ в REPL во время эксплуатации

В разделе «Постоянное изменяющееся окружение» выше мы говорили о «постоянном и изменяющемся окружении», предоставляемом оболочкой REPL в контексте разработки. Однако там не говорилось, что область применения REPL не ограничивается этапом разработки. Нет никаких технических причин, препятствующих использованию REPL в контексте эксплуатации, чтобы получить те же преимущества динамизма, что и во время разработки.

Не забывайте, что основной единицей взаимодействий между вами и средой Clojure является *загрузка кода*: просто выполняйте выражения из файлов на диске или введенные в REPL вручную, которые

определяют функции и значения и тем самым изменяют окружение. Никаких особых требований при этом не накладывается. Некоторые оболочки REPL привязаны к локальным процессам операционной системы (включая оболочку REPL по умолчанию, входящую в дистрибутив Clojure и представленную в примере 1.1), тем не менее, это не является универсальным правилом. В действительности многие (если не большинство) инструментов разработки программ на языке Clojure – включая Counterclockwise для Eclipse и SLIME для Emacs – используют только «удаленные» процессы REPL.

Когда оболочка REPL запускается с помощью этих инструментов, порождается новый процесс JVM, как того и следовало ожидать. Но этот процесс запускает сервер REPL, к которому подключается инструмент: весь код, который загружается из таких инструментов, и все действия, которые вы производите с оболочкой REPL с помощью их пользовательских интерфейсов, передаются через сетевой канал, хотя оба его конца связаны с локальным компьютером. Эти серверы REPL принимают отправленный код в виде текста и выполняют ту же последовательность операций *чтение*, *вычисление*, *вывод*, как и любая другая оболочка REPL, при этом выводимые результаты отправляются обратно в пользовательский интерфейс REPL инструмента, а не выводятся в `*out*`.

> **Примечание.** Серверы Clojure REPL, используемые обоими инструментами, Counterclockwise и SLIME, – nREPL и `swank-clojure`, соответственно – легко можно интегрировать в ваши приложения, чтобы получить возможность открывать сеанс REPL с ними после их развертывания и ввода в эксплуатацию. Оба используют библиотеки Clojure, доступные в виде зависимостей Maven. подробности о встраивании серверов можно узнать в документации по адресам: http://github.com/clojure/tools.nrepl и https://github.com/technomancy/swank-clojure.

Проще говоря, если вы сможете заставить приложение запустить сервер REPL и подключить к нему свой инструмент для работы с Clojure, вы сможете взаимодействовать с удаленным процессом Clojure, используя его оболочку REPL, как если бы он выполнялся на локальном компьютере. Однако, следует заметить, что кроме особых случаев не следует использовать возможность подключения к REPL развернутых, удаленных приложений для разработки новых возможностей. Подключение к REPL во время эксплуатации приложений предпочтительнее использовать для мониторинга, отладки и, иногда, внесения исправлений.

Мониторинг и анализ в процессе выполнения. Благодаря тому, что REPL поддерживает столь же надежное соединение с удаленным окружением, как и с локальным, вы можете использовать эту возможность для мониторинга данных и событий во время выполнения, что исключено с применением других средств. Например, возможность перехвата ключевых данных событий, их анализа и изменения в REPL открывает широкие горизонты. Ниже приводится пара тривиальных функций, дающих такую возможность:

```
(let [log-capacity 5000                                    ❶
      events (agent [])]                                   ❷
  (defn log-event [e]
    (send events #(if (== log-capacity (count %))
                    (-> % (conj e) (subvec 1))             ❸
                    (conj % e)))
    e)                                                     ❹
(defn events [] @events))                                  ❺
```

❶ Здесь определяется емкость (capacity) нашего «журнала». В данном случае значение размера жестко «зашито» в программный код, но совсем несложно оформить его в виде конфигурационного параметра.

❷ Роль «журнала» играет простой агент, инициализированный пустым вектором. Агент используется, чтобы иметь возможность изменять вектор журнала, не блокируя выполнение потока, вызвавшего `log-event`, как это происходит, например, при использовании атомов.

❸ Операция записи события в «журнал» просто добавляет его в конец вектора. Если вектор заполнен до предела, то перед добавлением нового события из вектора удаляется первый элемент.

❹ Функция `log-event` возвращает зарегистрированное событие, поэтому ее легко можно включать в формы потоковых макросов или комбинировать с другими функциями.

❺ Это простая функция доступа, позволяющая извлекать вектор с событиями из замыкания.

Посмотрим, как на практике можно использовать эту поддержку журналирования событий. Представьте, что имеется некоторое веб-приложение и вам потребовалось отследить, откуда поступает входящий трафик, чтобы, например, выявить новые «активные» домены, ссылающиеся на вас. Для этого можно было бы организовать перехват части данных запросов с помощью `log-event`; мы подготовили некоторые фиктивные данные для иллюстрации, имитирующие ссылки из нескольких доменов:

```
(doseq [request (repeatedly 10000 (partial rand-nth [{:referrer "twitter.com"}
                                                     {:referrer "facebook.com"}
                                                     {:referrer "twitter.com"}
                                                     {:referrer "reddit.com"}]))]
  (log-event request))
;= nil
(count (events))
;= 5000
```

Функция `log-event` работает правильно и сохраняет не более 5000 событий; причем первые 5000 начинают сохраняться с головы вектора. Если теперь подключиться к удаленному веб-приложению через REPL, с этими данными можно будет сделать *все, что угодно*; здесь мы выполняем простое суммирование, позволяющее нам увидеть, какие домены создали наибольший трафик к нам, судя по последним 5000 запросов:

```
(frequencies (events))
;= {{:referrer "twitter.com"} 2502,
    {:referrer "facebook.com"} 1280,
    {:referrer "reddit.com"} 1218}
```

Анализ последних 5000 запросов показал, что наибольший трафик создал домен Twitter (за счет удвоения количества «запросов» из домена Twitter в фиктивных данных). Конечно, можно было бы реализовать более обширный анализ данных, например, можно было бы сохранять не только информацию о домене, сославшемся на нас, и ограничивать вектор не количеством запросов а временем, чтобы, скажем, хранить в нем информацию о запросах, поступивших в течение последних 10 минут.

В любом случае, благодаря поддержке REPL мы получаем возможность внедряться в удаленные приложения на языке Clojure, и можем пользоваться всеми его средствами для извлечения, изменения и анализа состояния приложения. Обычно для решения такого рода задач требуется сохранять данные на диске или в базе данных и извлекать их оттуда для анализа или встраивать в приложения административные панели и другие индикаторы состояния, но все это требует дополнительных ресурсов и планирования, что иногда затруднительно или невозможно. Как бы то ни было, не так много инструментов, превосходящих REPL по удобству, и позволяющих немедленно выполнять любой анализ.

Исправление. После обнаружения ошибки и определения способа ее исправления (например, при проверке в некоторой среде тестирования), внесение исправлений в действующее приложение без его остановки, может оказаться невозможным из-за специфических особенностей приложения и применяемой локальной политики. Однако с помощью удаленного сеанса REPL внести исправления в работающее приложение очень просто, достаточно лишь загрузить измененный код.

Однако внесение исправлений требует некоторой осторожности и подготовки. В частности, существуют некоторые ограничения на код, который можно обновить в REPL, о чем подробнее рассказывается в разделе «Ограничения при переопределении конструкций» ниже. Кроме того, существуют проблемы, связанные с логистикой: если для исправления ошибки потребовалось внести изменения в несколько файлов, их необходимо загружать поочередно и в правильном порядке. Это обусловлено тем, что оболочка REPL, к которой вы подключаетесь, хранит свой список путей classpath, где хранится еще не измененный, устаревший код – объявления `require` и `use` не смогут как по волшебству найти измененный код на вашей машине.

Помимо этих проблем, вынуждающих проявлять осторожность при исправлении ошибок в развернутых приложениях, бывают ситуации, когда приходится часто изменять и обновлять «живую» среду. Иногда требования пользователей меняются настолько быстро, а время на выработку решений является настолько критичным, что имеет смысл интерпретировать «промышленную» среду как среду разработки[1]. В таких ситуациях возможность внедрения в развернутое приложение с помощью REPL и быстрой загрузки в него нового кода сложно проигнорировать.

Особые замечания по поводу «развертываемых» оболочек REPL

Если наличие интерактивного соединения с приложением, развернутым в эксплуатационном окружении, вызывает тревогу, мы можем напомнить вам о существовании в течение уже многих лет

---

[1] Такое действительно бывает в некоторых окружениях. Типичным примером может служить контекст «приемочных испытаний, проводимых пользователем», где, чем быстрее вы сможете внедрить улучшения, тем большее чувство удовлетворения будут испытывать ваши клиенты и партнеры.

механизма JMX (Java Management Extensions – расширения Java для управления), широко используемого для внесения изменений в выполняющиеся Java-приложения. И, тем не менее, есть несколько замечаний, которые необходимо учитывать при использовании REPL в эксплуатационном окружении.

Все изменения носят временный характер. Как и во время разработки, любые изменения в развернутом приложении носят исключительно временный характер. Загрузка кода или изменение структур данных через REPL влияют только на текущий процесс JVM/Clojure, но все эти изменения пропадут, если, к примеру, перезапустить развернутое приложение. В контексте разработки, напротив, изменения сначала производятся в файлах с исходным кодом, и только потом загружается измененный код. Когда в следующий раз проект будет запущен в оболочке REPL, будет загружен уже измененный код.

Об этом нужно либо просто помнить, работая в оболочке REPL, подключенной к удаленному приложению, или предусматривать пути решения проблемы. Одним из таких путей может быть обеспечение передачи в удаленную среду новых артефактов или выполняемых файлов с любым новым кодом, загружаемым в приложение, чтобы при перезапуске приложения, изменения, загруженные динамически через REPL, не потерялись и всегда присутствовали в *.war* или *.jar*-файлах.

Сетевая безопасность и контроль доступа. Ни одна из реализаций сетевых оболочек Clojure REPL не поддерживает обеспечение безопасности посредством аутентификации или шифрования трафика. Обычно все это обеспечивается независимыми средствами.

Проще всего решить эту проблему, защитив сетевой порт, к которому подключен сервер REPL приложения, брандмауэром системы[1]. Для доступа к удаленной системе можно также создать SSH-туннель или использовать VPN и подключить сервер REPL через безопасный шлюз.

Также серверы REPL не поддерживают какие-либо механизмы управления доступом: установив соединение с оболочкой REPL, вы получаете неограниченный доступ к среде выполнения Clojure и к окружению, в котором она находится. С точки зрения безопас-

---

[1] Весьма вероятно, что это будет сделано по умолчанию, так как в промышленных системах открываются только порты общедоступных сетевых служб, таких как HTTP, HTTPS и SSH.

ности, соединение с REPL следует защищать сеансом SSH. Тем не менее, существуют дополнительные способы ограничения возможностей REPL за счет настройки изолированной среды («песочницы») вычислений[1].

Ограничения при переопределении конструкций

Возможность в интерактивном режиме переопределять части приложения на языке Clojure почти ничем не ограничивается. Все, что имеется в программе, от функций и структур данных верхнего уровня, до типов, объявленных с помощью `deftype` и `defrecord`, протоколов и мультиметодов, можно переопределить и изменить во время выполнения (разумеется, в пределах каждой конструкции), просто загрузив новый или измененный код. При этом оговорка «почти» относится только к ограничениям самой виртуальной машины JVM.

Изменение списка путей classpath. Вообще список путей classpath в JVM нельзя изменить или дополнить во время выполнения. Это означает, что нельзя загрузить и использовать новые зависимости, недоступные процессу JVM/Clojure на момент его запуска. Однако сообществами Java и Clojure было выработано множество обходных решений[2].

Форма gen-class не обладает динамическими возможностями. Как описывается в разделе «Определение именованных классов», в главе 9, форма `gen-class` генерирует статические классы Java, причем, *только* на этапе предварительной компиляции. Такие классы по определению не могут обновляться во время выполнения. В экосистеме JVM были выработаны некоторые решения, позволяющие перезагружать и обновлять статические классы (к которым относятся и сгенерированные с помощью `gen-class`), но знакомство с ними далеко выходит за рамки этой книги.

---

[1] Одной из реализаций такой ограниченной среды («песочницы») является *clojail* – https://github.com/flatland/clojail – которая проверялась в «боевых» условиях, путем вычисления фрагментов программного кода в различных каналах Clojure IRC и на сайте 4clojure.com.

[2] Одним из них является библиотека *pomegranate* (https://github.com/cemerick/pomegranate), позволяющая добавлять *.jar*-файлы в classpath виртуальной машины JVM, либо непосредственно с диска, либо через разрешение зависимостей Maven.

Экземпляры классов навсегда сохраняют встроенные реализации. Встроенные (inline) реализации интерфейсов и протоколов в классах, объявленных с помощью `deftype` и `defrecord`, не могут изменяться динамически в существующих экземплярах этих классов. Одним из обходных решений этой проблемы является использование приема делегирования, позволяющего отделить функции, которые можно будет переопределить при необходимости (именно такой подход был предпринят при реализации `ActionListener` в примере 10.1), а вместо передачи функций в виде именованных переменных при реализации протоколов с помощью макроса `extend`, передавать сами переменные, используя нотацию `#'`. Когда позднее функция изменится, реализация протокола будет использовать новую функцию.

Кроме этих ограничений, накладываемых виртуальной машиной JVM, имеется также пара ограничений, свойственных самому языку Clojure:

Переопределение макросов не приводит к повторному их применению. Если вы определили макрос, и использовали его в определении функции, при переопределении макроса изменится только сам макрос, а определение функции не изменится. Помните, что макросы применяются только на этапе компиляции: чтобы «применить» новую реализацию макроса, необходимо перезагрузить (и, соответственно, перекомпилировать) весь код, где он используется.

Переопределение мультиметода не влечет за собой изменение функцию выбора (dispatch function). Как указывалось в примечании «Переопределение мультиметода не влечет за собой изменение функции выбора» в главе 7, макрос `defmulti` имеет семантику `defonce`, поэтому функции выбора не изменяются простой загрузкой измененной формы `defmulti`. Обходное решение заключается в применении функции `ns-unmap` к переменной мультиметода, что, к сожалению, требует перезагрузки реализаций всех методов мультиметода.

Различие ситуаций, когда выполняется обращение к значению переменной, а когда ее разыменование. Не забывайте, что форма `def` и родственные ей создают переменные в текущем пространстве имен, а переменные содержат фактические значения. Символ переменной вычисляется в значение переменной, имевшее место на момент обращения. Это в точности соответствует вашим ожиданиям, если, например, вы выполняете простой вызов функции. Однако,

при передаче имени переменной в виде аргумента другой функции, вы передаете значение этой переменной, а не саму переменную. То есть, после переопределения переменной по-прежнему будет использоваться ее старое значение:

```
(defn a [b] (+ 5 b))
;= #'user/a
(def b (partial a 5))      ❶
;= #'user/b
(b)
;= 10
(defn a [b] (+ 10 b))      ❷
;= #'user/a
(b)                        ❸
;= 10
```

❶ Мы определяем b как функцию, к функции в переменной a применяется единственный аргумент, в результате чего создается частично примененная функция.
❷ Переопределяем a...
❸ ...но так как b хранит оригинальную функцию a, переопределение не оказывает желаемого влияния.

Чтобы решить эту проблему, необходимо передать в аргументе саму переменную a:

```
(def b (partial #'a 5))    ❶
;= #'user/b
(b)
;= 15
(defn a [b] (+ 5 b))       ❷
;= #'user/a
(b)                        ❸
;= 10
```

❶ Мы передали форме partial аргумент #'a, вместо a; в результате возвращается частично примененная функция, хранящая ссылку на переменную a, а не ее значение, имевшее место на этот момент.
❷ Переопределим a...
❸ ... и теперь b будет использовать переопределенную версию функции, хранящуюся по ссылке #'a.

В заключение

Интерактивная оболочка Clojure REPL – это инструмент, который может превратить программирование в увлекательнейшее занятие, ускорить обнаружение причин, вызывающих ошибки, и помочь победить проблемы, обнаруживаемые в эксплуатационной среде. Эффективное ее применение является залогом получения максимальной выгоды от использования любых инструментов поддержки Clojure, а полное понимание ее потенциала является неотъемлемой частью опыта программирования на языке Clojure.

Часть IV
ПРАКТИКУМ

Глава 11. Числовые типы и арифметика

При разработке многих классов приложений программисты могут оставаться в счастливом неведении о тонкостях и особенностях реализации арифметики, независимо от языка или используемой среды выполнения. Узкими местами в таких приложениях обычно являются ввод/вывод, операции с базами данных и другие факторы.

Однако в некоторых областях производительность операций с числами и/или точность вычислений являются чрезвычайно важными факторами и, похоже, что количество этих областей только увеличивается со временем: крупномасштабная обработка данных, визуализация, статистический анализ и другие подобные классы приложений часто требуют значительной математической строгости. Clojure дает возможность выбирать способы оптимизации использования числовых данных в приложениях, чтобы удовлетворить этим двум основным требованиям. Не жертвуя краткостью, выразительностью или динамизмом во время выполнения, вы можете выбрать для использования:

1. Простые числовые типы, чтобы получить максимальную производительность.
2. Упакованные числовые типы (boxed numerics), чтобы обеспечив выполнение операций с произвольной точностью.

В этой главе мы исследуем модель представления чисел в языке Clojure и реализации операций над ними.

Числовые типы в Clojure

Прежде всего нам необходимо разобраться с основами – представлением числовых типов, поддерживаемых языком Clojure и перечисленных в табл. 11.1[1]. Для начала было бы хорошо сравнить

---

[1] Обращайтесь к табл. 1.2, чтобы вспомнить синтаксис определения литералов каждого числового типа.

числовые типы в Clojure с числовыми типами в Ruby и Python (те, кто знаком с Java, не должны испытывать сложностей, потому что Clojure использует модель представления чисел языка Java).

Таблица 11.1. Сравнение числовых типов в языке Clojure с числовыми типами в Ruby и Python

Числовой тип	Представление в Clojure	Эквивалент в Python	Эквивалент в Ruby
Простые 64-битные целые	long	Отсутствуют. В Python и Ruby отсутствует поддержка простых числовых типов (в обоих языках все числовые типы являются упакованными)	
Простые 64-битные вещественные десятичные IEEE	double		
Упакованные целые	java.lang.Long	`int`[a]	`Fixnum`[b]
Упакованные вещественные	java.lang.Double	float	Float
«Большие» целые (целые числа неограниченной точности)	java.math.BigInteger и clojure.lang.BigInt	long	Bignum
«Большие» вещественные (вещественные числа неограниченной точности)	java.math.BigDecimal	decimal.Decimal	BigDecimal
Рациональные числа (иногда называют отношениями или пропорциями)	clojure.lang.Ratio	fractions.Fraction	Rational

[a] Тип `int` в Python имеет размер 32 бита и потому очень близок к типу `java.lang.Integer` в JVM, однако, как отмечается в следующем разделе, все простые числовые типы в Clojure имеют размер 64 бита и потому соответствуют типу `long` (или `Long`, в данном случае).

[b] Размер типа `Fixnum` в Ruby зависит от реализации и аппаратной платформы, и обычно составляет 31 или 63 бита (последний бит зарезервирован для реализации семантики типа `Fixnum`; краткое обсуждение этого типа можно найти в разделе «Идентичность объектов (identical?)», ниже).

Давайте познакомимся с терминологией и поближе рассмотрим некоторые ключевые моменты из табл. 11.1.

В Clojure предпочтение отдается 64-битным (или больше) представлениям

В JVM поддерживаются числовые типы с более узким диапазоном представления, такие как 32-битные вещественные, 32- и 16-битные целые (`float`, `int` и `short`, соответственно), однако все формы механизма чтения и операции с числами в языке Clojure производят числа в 64-битном (или больше) представлении. Все арифметические операции могут принимать числовые типы с более узким диапазоном представления, но в Clojure они всегда возвращают значения, соответствующие 64-битным представлениям. Например, операция увеличения 32-битного целого вернет 64-битное целое:

```
(class (inc (Integer. 5)))
;= java.lang.Long
```

Это делает числовую модель в Clojure более простой, чем в Java, где существует три разных представления целых чисел и два разных представления вещественных чисел.

Clojure имеет смешанную модель числовых типов

В отличие от большинства динамических языков (включая Ruby и Python), наряду с упакованными числовыми типами в языке Clojure поддерживаются простые числовые типы (например, `long` и `double`).

Простые числовые типы не являются объектами – они являются типами значений и непосредственно соответствуют машинным числовым типам, арифметические операции над которыми выполняются на аппаратном уровне. В Clojure используются простые типы `long` и `double` из JVM, которые соответствуют типам `long long` (или `int64_t`) и `double` в языке C/C++, соответственно.

Упакованные числовые типы (boxed numbers), напротив, *являются объектами*, которые определяются классами; `java.lang.Long` – это класс-обертка, единственное назначение которого состоит в том, чтобы хранить значение простого типа `long`, а `java.lang.Double` – это, соответственно, класс, содержащий значение простого типа `double`[1].

[1] Простые типы всегда обозначаются именами, содержащими только буквы нижнего регистра (такими как `double`), тогда как упакованные представления всегда идентифицируются именами классов, начинающимися с буквы верхнего регистра (такими как `Double`).

Как объекты, применение этих типов влечет за собой дополнительные накладные расходы, из-за чего операции с ними выполняются медленнее: перед началом операции обычно требуется распаковать объект they (чтобы получить значение простого типа), а результат вновь упаковать в объект (что требует создания нового экземпляра ~~класса~~, соответствующего типу значения результата).

Зачем вообще нужны типы Long и Double, если они не дают никаких семантических преимуществ перед родственными простыми типами?

При рассмотрении в отдельности создается впечатление, что классы-обертки `Long` и `Double` ни на что не годятся: их использование в арифметических операциях влечет дополнительные накладные расходы, и они не дают дополнительных преимуществ в виде более широкого диапазона представления чисел или более высокой точности, как `BigInteger` и `BigDecimal`. Причина существования классов `Long` и `Double` (и других классов-оберток в Java, соответствующих простым типам в JVM, таких как `Boolean` и `Short`) в том, что они позволяют использовать числа там, где полезнее использовать объекты.

Например, без этих классов-оберток было бы невозможно хранить числа в ассоциативных массивах и других коллекциях. Java Collections API работает исключительно с объектами.

Мы еще вернемся к обсуждению упакованных числовых типов, в частности, когда будем рассматривать, какое влияние они оказывают на Clojure в разделе «Оптимизация производительности операций с числами» ниже.

Благодаря прямому соответствию между простыми и машинными типами, и из-за накладных расходов, присущих упакованным числовым типам, операции с простыми типами всегда выполняются быстрее – иногда на порядки быстрее, в зависимости от особенностей алгоритма. С другой стороны, 64-битные числовые типы, поддерживаемые виртуальной машиной JVM, имеют ограничения на диапазон и точность представления. Чтобы восполнить этот недостаток, в Clojure используются типы `BigDecimal` и `BigInteger` – два числовых типа из Java, обеспечивающие неограниченные диапазон и точность представления чисел – и реализован собственный тип `BigInt`. Эти типы определены в виде классов и потому им свойственны те же накладные расходы, что и для упакованных числовых типов, зато они позволяют работать с числами произвольной величины и с произвольной точностью.

Итак, у нас есть два способа разделения числовых типов в Clojure: по представлению – *простые* или *упакованные*, и по наличию или отсутствию ограничений на точность или диапазон. Различные конкретные представления целых и вещественных чисел, разделенные этими двумя способами, показаны в виде матрицы в табл. 11.2.

Таблица 11.2. Сравнительная матрица числовых типов в Clojure

	Ограниченные по диапазону/точности	**Неограниченные по диапазону/точности**
Простые типы	long, double	Нет
Объектные типы	java.lang.Long, java.lang.Double	clojure.lang.BigInt, java.math.BigDecimal, java.math.BigInteger

Несмотря на то, что в Clojure поддерживается множество различных числовых типов, семантика арифметических операций не зависит от конкретных типов значений, принимающих участие в операциях. Например, `dec` всегда уменьшает свой аргумент, независимо от его типа, и всегда возвращает число того же конкретного типа:

```
(dec 1)
;= 0
(dec 1.0)
;= 0.0
(dec 1N)
;= 0N
(dec 1M)
;= 0M
(dec 5/4)
;= 1/4
```

Аналогично мы свободно можем смешивать различные числовые типы в одной операции[1]:

```
(* 3 0.08 1/4 6N 1.2M)
;= 0.432
(< 1 1.6 7/3 9N 14e9000M)
;= true
```

[1] Приятное усовершенствование, в сравнении с Java, особенно если учесть, что арифметические операторы в Java не могут использоваться для выполнения операций с числовыми типами, обеспечивающими произвольную точность.

Если аргументы арифметической операции имеют разные типы, тип результата определяется правилами преимущества более широкого типа. Эти правила обсуждаются в разделе «Правила определения типа результата» ниже.

Рациональные числа

Рациональные числа – это множество чисел, которые можно выразить дробью двух целых чисел. Например, $^1/_3$ и $^3/_5$ являются рациональными числами (они равны `0.333...` и `0.6`, соответственно). Большинство языков программирования – включая Java, Ruby и Python – поддерживают только целые и вещественные числа и операции с ними. Поэтому, когда встречается рациональное число, оно немедленно «сплющивается» в приближенное представление с плавающей точкой:

```
# Ruby
>> 1.0/3.0
0.333333333333333

# Python
>>> 1.0/3.0
0.33333333333333331
```

Опасности использования представлений с плавающей точкой в различных вычислениях широко известны (хотя мало кто хорошо понимает их). Ниже представлен типичный пример на языке Clojure:

```
(+ 0.1 0.1 0.1)
;= 0.30000000000000004
```

Ошибка, вкравшаяся в результат, является простым следствием особенностей представления вещественных чисел[1]. Clojure позволя-

[1] Как и во многих других средах выполнения, вещественные числа в JVM представляются в соответствии со спецификацией IEEE 754. Если вам интересно узнать, как хранятся вещественные числа в памяти, в виде набора битов и байтов, обращайтесь к описанию спецификации по адресу: http://en.wikipedia.org/wiki/IEEE_754-2008. (Для русскоязычных читателей можно порекомендовать перевод стандарта на русский язык: http://www.softelectro.ru/ieee754.html. – *Прим. перев.*)

ет избежать этого (а) поддерживая литералы рациональных чисел, и (б) не преобразуя рациональные числа в менее точное представление с плавающей точкой:

```
(+ 1/10 1/10 1/10)
;= 3/10
```

Оборотной стороной этой особенности является возможность преобразования рациональных чисел в целые без потери точности:

```
(+ 7/10 1/10 1/10 1/10)
;= 1
```

Рациональные числа могут явно преобразовываться в вещественные:

```
(double 1/3)
;= 0.3333333333333333
```

а вещественные числа могут преобразовываться в рациональные, с помощью функции `rationalize`:

```
(rationalize 0.45)
;= 9/20
```

Правила определения типа результата

Когда в арифметической операции принимают участие значения разных числовых типов, тип возвращаемого значения определяется с использованием фиксированной иерархии. Все числовые типы имеют разную *ширину*, и аргумент операции с самым широким типом определяет тип возвращаемого значения (см. рис. 11.1).

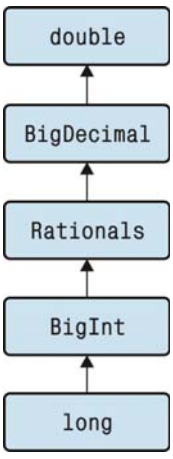

Рис. 11.1. Числовые типы в Clojure образуют иерархию в соответствии с их шириной

Любая арифметическая операция должна вернуть значение определенного типа, и, соответственно, должна иметь возможность выяснить, какой тип выбрать, когда в операцию вовлечены аргументы

разных типов. Типы в иерархии, определенной в Clojure и изображенной на рис. 11.1, упорядочены так, чтобы приведение значения к типу результата всегда выполнялось без «потерь». Например, любое значение типа `long` можно привести к длинному целому, рациональному или вещественному числу без потери значимых разрядов, однако обратное утверждение неверно.

Это легко продемонстрировать:

```
(+ 1 1)          ❶
;= 2
(+ 1 1.5)        ❷
;= 2.5
(+ 1 1N)         ❸
;= 2N
(+ 1.1M 1N)      ❹
;= 2.1M
```

❶ Операции с однотипными аргументами возвращают результат того же типа.

❷ Операция над значениями `long` и `double` всегда будет возвращать `double`.

❸ Операция над значениями `long` и `BigInteger` всегда будет возвращать `BigInteger`.

❹ Операция над значениями `BigDecimal` и `BigInteger` всегда будет возвращать `BigDecimal`.

Единственная проблема в том, что любая операция с участием значения типа `double` всегда будет возвращать значение типа `double`, даже при том, что тип `double` не может корректно представить полный диапазон значений других числовых типов. Это обусловлено тем, что:

1. Тип `double` (следующий стандарту IEEE) определяет некоторые специальные значения, которые не могут быть представлены типом `BigDecimal` (в частности, `Infinity` и `NaN`).
2. Тип `double` является единственным, страдающим врожденной неточностью – было бы странным из операции, выполняемой над неточным числом, возвращать значение типа, подразумевающего точность.

Понятие ширины типа распространяется не только на арифметические операции. Эти операции являются обычными функциями, поэтому те же самые правила применяются и к функциям, принимающим числовые аргументы и реализованные с применением опера-

торов Clojure. Рассмотрим простую функцию, вычисляющую сумму квадратов:

```
(defn squares-sum
  [& vals]
  (reduce + (map * vals vals)))
;= #'user/squares-sum
(squares-sum 1 4 10)
;= 117
```

Что произойдет, если к суммируемым значениям добавить значение типа `double`, `BigInt`, `BigDecimal` или `Ratio`? Мы получим результат типа `double`, `BigInt`, `BigDecimal` или `Ratio`, соответственно, независимо от типов других суммируемых значений, в соответствии с иерархией, изображенной на рис. 11.1:

```
(squares-sum 1 4 10 20.5)
;= 537.25
(squares-sum 1 4 10 9N)
;= 198N
(squares-sum 1 4 10 9N 5.6M)
;= 229.36M
(squares-sum 1 4 10 25/2)
;= 1093/4
```

Арифметика в Clojure

Знания числовых типов и особенностей их представления недостаточно для понимания числовой модели в Clojure. Различные арифметические операторы и операторы сравнения в языке Clojure предоставляют дополнительные семантические гарантии получения достоверных результатов для всех числовых типов и неподвержены многим типичным проблемам вычислительной математики, таким как переполнение (overflow) и потеря значимых разрядов (underflow), благодаря управлению типом результата операции и единообразию.

Ограниченная и произвольная точность

Диапазон значений 64-битных типов `long` и `double`, очень широк. Тип `long` позволяет представлять целые числа в диапазоне $\pm 2^{63}-1$, а тип `double` позволяет представлять вещественные числа в диапазоне $\pm 1.7976931348623157^{308}$. Большинство приложений не нуждаются

в более широком диапазоне или более высокой точности представления, поэтому зачастую типов `long` и `double` бывает вполне достаточно.

Для решения задач, где требуется более широкий диапазон и/или более высокая точность, предоставляются числовые типы, обеспечивающие *произвольную точность*. Эти представления реализованы в виде упакованных типов. Для представления целых чисел произвольной величины в Clojure используются два типа: `clojure.lang.BigInt` и `java.math.BigInteger`, а для представления вещественных чисел с произвольной точностью – тип `java.math.BigDecimal`.

Зачем в Clojure реализован свой класс BigInt, если в Java уже имеется класс BigInteger?

На то имеется две причины.

Во-первых, реализация `BigInteger` имеет некоторые недостатки. В частности, метод `.hashCode` этого класса не соответствует одноименному методу класса `Long`:

```
(.hashCode (BigInteger. "6948736584"))
;= -1641197977
(.hashCode (Long. 6948736584))
;= -1641198007
```

Это – большая проблема, способная привести к ситуации, когда два эквивалентных значения могут оказаться в одном множестве (или соответствовать двум разным значениям в ассоциативном массиве, например), просто из-за особенностей реализации конкретных типов[1].

Во-вторых, все операции над значениями типа `BigInteger` должны выполняться с использованием его (медленной) программной реализации, а Clojure оптимизирует арифметику с типом `BigInt`, используя аппаратные (более быстрые) операции всегда, когда это возможно, *при условии, что аргументы операции находятся в пределах диапазона представления простого 64-битного типа* `long`. Это означает, что платить высокую цену за произвольную точность часто приходится, только когда она действительно необходима.

Наконец, вам не следует волноваться о необходимости выбора между этими двумя целочисленными типами. Типы аргументов никак не влияют на семантику любых арифметических операций в Clojure, и независимо от конкретных типов аргументов они никогда не вернут значение типа `BigInteger`.

---

[1] Пример такой ситуации приводится в разделе «Эквивалентность может защитить ваш рассудок» ниже.

Чтобы приблизить все эти разговоры о точности поближе к практике, посмотрим, что можно делать с целыми числами произвольной величины, чего не позволяют обычные значения типа `long`. Рассмотрим некоторое значение в программе, которое по совпадению является максимально возможным значением, которое способен представить тип `long`:

```
(def k Long/MAX_VALUE)
;= #'user/k
k
;= 9223372036854775807
```

Это большое число, но иногда недостаточно большое:

```
(inc k)
;= ArithmeticException integer overflow
```

При целочисленном переполнении арифметические операторы Clojure возбуждают исключение, вместо того, чтобы по-тихому «выполнить перенос», как описывается в разделе «Неконтролируемые операции» ниже. Но иногда нам действительно бывает необходимо обрабатывать значения, не умещающиеся в диапазон представления `long` и `double`; в таких случаях у нас есть на выбор два варианта:

Явно использовать числовые типы произвольной точности. В зависимости от конкретных требований и контекста для этой цели можно использовать функции преобразования `bigint` и `bigdec`:

```
(inc (bigint k))
;= 9223372036854775808N
(* 100 (bigdec Double/MAX_VALUE))
;= 1.797693134862315700E+310M
```

или соответствующую форму записи литералов. В табл. 1.2 указывалось, что окончание `N` в литералах производит значение типа `BigInt`, а окончание `M` производит значение типа `BigDecimal`:

```
(dec 10223372636454715900N)
;= 10223372636454715899N
(* 0.5M 1e403M)
;= 5E+402M
```

Кроме того, литералы целых чисел, не укладывающихся в диапазон представлений 64-битного типа `long` автоматически производят значения типа `BigInt`:

```
10223372636454715900
;= 10223372636454715900N
(* 2 10223372636454715900)
;= 20446745272909431800N
```

Для операций с целыми числами использовать операторы, автоматически расширяющие тип. Явно использовать числовые типы с произвольной точностью можно, когда мы полностью контролируем входные значения некоторой функции; это позволяет использовать базовые арифметические операторы Clojure и полагаться на поддержку расширения числовых типов при возвращении результатов. Если известно, что в конкретных вычислениях потребуется использовать представление с неограниченной точностью или диапазоном величин, мы можем подавать на вход значения произвольной точности.

С другой стороны, при реализации вычислений или алгоритмов, в ходе выполнения которых может произойти выход за пределы диапазона представления целых чисел типа `long`, а механизма расширения типов недостаточно, чтобы гарантировать правильность результатов, можно использовать варианты арифметических операторов Clojure *со штрихом*[1], автоматически расширяющих тип `long` результата до типа `BigInt`, чтобы исключить вероятность переполнения:

```
(inc' k)
;= 9223372036854775808N
```

Однако эти операторы расширяют тип только в случае необходимости. Это гарантирует, что если результат умещается в границы диапазона 64-битного типа `long`, его тип не будет расширен:

---

[1] Здесь подразумеваются соглашения по использованию *штриха* (обычно изображается как апостроф, ') в качестве окончания в имени оператора, чтобы обозначить вариант базового оператора. Подробнее о значении штриха за пределами Clojure можно узнать по адресу: http://en.wikipedia.org/wiki/Prime_(symbol).

```
(inc' 1)
;= 2
(inc' (dec' Long/MAX_VALUE))
;= 9223372036854775807
```

Варианты со штрихом существуют для всех арифметических операторов Clojure, которые могут вызвать ошибку переполнения или потери значимых разрядов: `inc'`, `dec'`, `+'`, `-'` и `*'`.

Операторы со штрихом имеют свои, пусть и небольшие, накладные расходы; каждая операция с типами, имеющими ограниченное представление, должна проверять результат и может повториться для значений типа `BigInt` вместо `long`. Эти накладные расходы – цена гарантий получения правильного результата с минимумом усилий.

Неконтролируемые операции

Переполнение и *потеря значимых разрядов* – это такие ситуации, когда результат операции над целыми числами оказывается вне диапазона, поддерживаемого представлением целых чисел. Эффект переполнения и потери значимых разрядов наверняка знаком каждому, кому приходилось работать с числовыми данными в Java[1]; например, следующий Java-код:

```
System.out.println(Long.MAX_VALUE);
System.out.println(Long.MAX_VALUE + 1);
```

выведет следующее:

```
9223372036854775807
-9223372036854775808
```

Вот так так! Конечно, никто из нас никогда преднамеренно не пытался увеличить максимально возможное целое число, но то же самое происходит, когда результат вычислений (непреднамеренно) выходит за границы представления данного числового типа.

---

[1] Переполнение и потеря значимых разрядов не возникают в некоторых языках, таких как Python и Ruby, каждый из которых постоянно поддерживает включенным механизм автоматического расширения типов, чтобы избежать таких ошибок.

К счастью, все арифметические операторы Clojure проверяют возникновение ситуации переполнения и потери значимых разрядов, и при необходимости возбуждают исключение:

```
Long/MIN_VALUE
;= -9223372036854775808
(dec Long/MIN_VALUE)
;= #<ArithmeticException java.lang.ArithmeticException: integer overflow>
```

Это определенно лучше, чем пытаться отыскать причины странного поведения приложения из-за того, что какая-то операция вызвала переполнение и вернула семантически ошибочный результат.

Однако в некоторых редких случаях бывает желательно сохранить неконтролируемое поведение операторов Java, например, когда желательно:

1. Сохранить полную совместимость с семантикой операторов Java, например, если необходимо реализовать на Clojure новую версию некоторой функциональности, которая прежде была реализована на Java.
2. Избежать накладных расходов (очень небольших), связанных с проверками на переполнение/потерю значимых разрядов в языке Clojure.

Для всех операторов Clojure существуют их варианты `unchecked-*`, которые не выполняют проверку:

```
(unchecked-dec Long/MIN_VALUE)
;= 9223372036854775807
(unchecked-multiply 92233720368547758 1000)
;= -80
```

Имена этих вариантов слишком длинные (имя `unchecked-multiply` выглядит менее удобным в использовании, чем `*`). Однако есть и другая возможность: с помощью `set!` установить `*unchecked-math*` в значение `true` перед любой из форм верхнего уровня, которая должна выполнять неконтролируемые арифметические операции:

```
(inc Long/MAX_VALUE)
;= #<ArithmeticException java.lang.ArithmeticException: integer overflow>
(set! *unchecked-math* true)
;= true
(inc Long/MAX_VALUE)
```

```
;= -9223372036854775808
(set! *unchecked-math* false)
;= false
```

Внимание. При использовании неконтролируемых арифметических операций общепринято помещать форму `set! *unchecked-math*` в начало файла с исходными текстами, перед формами, где должны выполняться неконтролируемые операции. При необходимости можно вернуть переменную `*unchecked-math*` в значение `false` после этих форм. Переменная `*unchecked-math*` сбрасывается в исходное состояние в начале каждого исходного файла в момент его загрузки, поэтому ее необходимо устанавливать в каждом файле; то есть, эту переменную нельзя установить один раз в пространстве имен верхнего уровня и полагать, что эта настройка будет распространяться на все последующие загружаемые файлы. Если бы такое поведение имело место, то забыв сбросить `*unchecked-math*` обратно в значение `false` в конце файла, вы могли бы допустить «распространение» нежелательной семантики неконтролируемых операций на всю программу, которая может включать не только ваш код.

Обратите внимание, что обычно нельзя использовать `binding` для изменения значения `*unchecked-math*`:

```
(binding [*unchecked-math* true]
(inc Long/MAX_VALUE))
;= #<ArithmeticException java.lang.ArithmeticException: integer overflow>
```

Это объясняется тем, что `*unchecked-math*` управляет действиями компилятора, а форма `binding` не оказывает никакого эффекта до этапа выполнения. Она выполняется после того, как компилятор определится с выбором операций[1].

Режимы масштабирования и округления в операциях с вещественными числами произвольной точности

Одной из неприятных сторон использования типа `BigDecimal` в Java является то, что многие операции с ним по умолчанию вызывают ошибки:

[1] Выражение, передаваемое функции `eval` внутри такой формы `binding`, можно было бы скомпилировать и вычислить с применением неконтролируемых операций, потому что компиляция будет выполнена после создания динамической области видимости... однако нам трудно представить, где это может пригодиться.

```
new BigDecimal(1).divide(new BigDecimal(3));

= java.lang.ArithmeticException:
= Non-terminating decimal expansion; no exact representable decimal result.
```

Этот код итак слишком длинный, а если еще добавить инструкцию, определяющую режим округления и максимальный масштаб, он будет выглядеть еще хуже:

```
new BigDecimal(1).divide(new BigDecimal(3),
                         new MathContext(10, RoundingMode.HALF_UP));

= 0.3333333333
```

Реализация вещественных чисел произвольной точности в Ruby требует заплатить почти такую же цену, хотя и несколько иначе: вместо того, чтобы в каждой операции указывать математический контекст, параметры контекста устанавливаются глобально, в классе `Decimal`.

В Clojure используется иной подход: он предоставляет макрос `with-precision`, использование которого существенно упрощает программный код. Вы указываете желаемый масштаб (и режим округления, если необходимо) и эта информация автоматически передается во все операции с типом `BigDecimal`, выполняемые в области видимости `with-precision`:

```
(/ 22M 7)
;= #<ArithmeticException java.lang.ArithmeticException:
;=   Non-terminating decimal expansion; no exact representable decimal
;=   result.>
(with-precision 10 (/ 22M 7))
;= 3.142857143M
(with-precision 10 :rounding FLOOR
  (/ 22M 7))
;= 3.142857142M
```

А в случае действительной необходимости, масштаб и режим округления можно определить без использования макроса `with-pre-`

cision, сохранив в переменной *math-context* соответствующий экземпляр класса java.math.MathContext[1]:

```
(set! *math-context*
       (java.math.MathContext. 10 java.math.RoundingMode/FLOOR))
;= #<MathContext precision=10 roundingMode=FLOOR>
(/ 22M 7)
;= 3.142857142M
```

Равенство и эквивалентность

Язык Clojure поддерживает три способа определения равенства значений, реализованные в виде трех различных функций-предикатов.

Идентичность объектов (identical?)

Определение идентичности объектов, выполняется с помощью функции identical? и позволяет выяснить, являются ли два (или более) объекта одним и тем же экземпляром. Эта функция является прямым аналогом оператора == в Java (когда последний используется для сравнения ссылок на объекты), оператора is в Python и функции equal? в Ruby:

```
(identical? "foot" (str "fo" "ot"))
;= false
(let [a (range 10)]
  (identical? a a))
;= true
```

В общем случае одинаковые числа никогда не будут идентичны друг другу, даже в виде литералов:

---

[1] Поскольку *math-context* является динамической переменной, вы можете изменять ее значение либо с помощью set!, либо alter-var-root. Выбор зависит от того, желаете ли вы ограничить действие значения этой переменной определенным потоком выполнения или оно должно быть доступно глобально. Подробнее о динамических областях видимости рассказывается в разделе «Динамическая область видимости» в главе 4.

```
(identical? 5/4 (+ 3/4 1/2))
;= false
(identical? 5.4321 5.4321)
;= false
(identical? 2600 2600)
;= false
```

Исключение составляют так называемые *фиксированные числа* (fixnums)? поддерживаемые виртуальной машиной JVM (и, соответственно, языком Clojure). Фиксированные числа (fixnums) – это ограниченное множество упакованных целочисленных значений, используемые всегда, когда требуется распределить память для нового целого числа. В Ruby семантика фиксированных чисел охватывает весь диапазон целых чисел, а в Python диапазон фиксированных чисел включает в себя только значения от –5 до 256[1]. В Oracle JVM фиксированные числа занимают диапазон ±127[2], поэтому только для целочисленных значений из этого диапазона `identical?` будет возвращать истинное значение:

```
(identical? 127 (dec 128))
;= true
(identical? 128 (dec 129))
;= false
```

Вообще старайтесь не использовать функцию `identical?` для сравнения чисел.

Равенство ссылок (=)

Это то, что чаще всего подразумевают под словом «равенство»: чувствительное к типу (потенциально) глубокое сравнение значений с целью выявить их структурное сходство. Предикат `=` в Clojure заимствует семантику равенства ссылок из Java, определяемую ме-

---

[1] Такой необычный диапазон объясняется особенностями реализации CPython.

[2] Семантика фиксированных чисел в JVM определена как часть семантики преобразования простых значений в упакованные (описывается в §5.1.7 спецификации языка Java Language Spec (http://docs.oracle.com/javase/specs/ – *прим. перев.*)) и может отличаться между версиями и реализациями.

тодом `java.lang.Object.equals`[1]. Оператор `=` в Clojure функционально эквивалентен оператору `==` в Python и Ruby.

Он дает понятные результаты, согласующиеся с нашими представлениями, особенно когда сравниваются коллекции различных конкретных типов:

```
(= {:a 1 :b ["hi"]}
   (into (sorted-map) [[:b ["hi"]] [:a 1]])
   (doto (java.util.HashMap.)
     (.put :a 1)
     (.put :b ["hi"])))
;= true
```

Заметьте, однако, что коллекции из разных категорий никогда не будут признаны равными оператором `=`. Например, несмотря на то, что любая упорядоченная коллекция (вектор, список или последовательность) может быть равна другой упорядоченной коллекции другого конкретного типа, упорядоченная коллекция никогда не будет считаться равной множеству или ассоциативному массиву.

То же относится и к понятию равенства чисел. Если говорить точнее, оператор `=` вернет `true`, только когда сравниваемые числа относятся к одной категории, в противном случае он вернет `false`, даже если числа будут равными по величине. Например, все целочисленные типы могут благополучно сравниваться между собой с помощью оператора `=` (даже целочисленные типы более узкие, чем 64-битный тип `long` в Clojure), и вещественные числа с ограниченной точностью представления и различной ширины прекрасно сравниваются между собой:

```
(= 1 1N (Integer. 1) (Short. (short 1)) (Byte. (byte 1)))
;= true
(= 1.25 (Float. 1.25))
;= true
```

[1] Когда выполняется сравнение нечисловых значений, оператор `=` в Clojure просто вызывает `Object.equals` со своими аргументами. Специфика работы `Object.equals` в действительности гораздо сложнее, чем мы представили здесь. Вам стоит самим найти время и заглянуть в документацию с описанием этого метода, чтобы получить более полное представление о нем: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals%28java.lang.Object%29.

Однако оператор = *никогда* не вернет true при сравнении эквивалентных чисел из разных категорий:

```
(= 1 1.0)
;= false
(= 1N 1M)
;= false
(= 1.25 5/4)
;= false
```

В реализации оператора = *можно было бы* устранить эти различия (как это сделано в Ruby и Python), но это повлекло бы дополнительные накладные расходы и вызвало бы непонимание у тех, кому важна максимальная производительность программ, оперирующих однородными числовыми данными.

Вследствие этого было принято решение добавить в Clojure еще одно, третье понятие равенства, специально для тех, кому требуется сравнивать числа без учета их принадлежности к той или иной категории.

Числовая эквивалентность (==)

Предикат == в языке Clojure реализует понятие *числовой эквивалентности* в нашем интуитивном понимании, не отягощенном различиями между различными категориями, используемыми нами для представления чисел. Там, где оператор = возвращает false из-за сравнения чисел, принадлежащих разным категориям, оператор == может вернуть true, если сравниваемые числовые значения эквивалентны по величине:

```
(== 0.125 0.125M 1/8)
;= true
(== 4 4N 4.0 4.0M)
;= true
```

Обратите внимание, что оператор == может сравнивать даже рациональные числа с их десятичным представлением.

Оператор == требует, чтобы все его аргументы были числами, в противном случае он возбуждает исключение. Это означает, что если вы не уверены в том, какого типа аргументы будут сравниваться, следует использовать:

- оператор =;
- или, если требуется семантика определения эквивалентности, реализуемая оператором ==, необходимо проверять типы аргументов (например, с применением number?, как показано ниже).

```
(defn equiv?
  "Действует так же, как и оператор `==`, но не возбуждает исключение,
   если какой-то из аргументов не является числом."
  [& args]
  (and (every? number? args)
       (apply == args)))
;= #'user/equiv?
(equiv? "foo" 1)
;= false
(equiv? 4 4N 4.0 4.0M)
;= true
(equiv? 0.125 0.125M 1/8)
;= true
```

Эквивалентность может защитить ваш рассудок

Понятие равенства чисел в интерпретации языка Java действительно может доставлять неприятности при работе с коллекциями разнородных чисел. Поскольку реализации коллекций в Java зависят от определения равенства для каждого их члена (например, с целью выяснения принадлежности к множеству или для поиска элемента ассоциативного массива по указанному ключу), а результат сравнения чисел существенно зависит от их типов, вы можете столкнуться с такими неприятными ситуациями:

```
java.util.Map m = new java.util.HashMap();
m.put(1, "integer");
m.put(1L, "long");
m.put(java.math.BigInteger.valueOf(1), "bigint");
System.out.println(m);

>> {1=bigint, 1=long, 1=integer}
```

Вот так так! В ассоциативном массиве имеется три ключа, которые мы предпочли бы совместить в один. Эта особенность может приводить к невероятно трудноуловимым ошибкам. Коллекции в языке Clojure, напротив, применяют понятие эквивалентности к ключам ассоциативных массивов и членам множеств:

```
(into #{} [1 1N (Integer. 1) (Short. (short 1))])
;= #{1}
(into {}
       [[1 :long]
        [1N :bigint]
        [(Integer. 1) :integer]])
;= {1 :integer}
```

Остерегайтесь сравнений с участием вещественных чисел

Как обычно, работая с вещественными числами, необходимо учитывать особенности их реализации. Даже такие простые операции, как ниже, могут давать «неправильные» результаты:

```
(+ 0.1 0.2)
;= 0.30000000000000004
```

Операции сравнения вещественных чисел разных типов может проявлять неожиданное поведение. Взгляните:

```
(== 1.1 (float 1.1))
;= false
```

Обратите внимание, что результат `false` получен даже при том, что использовался предикат эквивалентности `==`, не чувствительный к типам своих операндов. Проблема в том, что тип `float` (32-битное представление вещественных чисел) потребовалось расширить до типа `double` (64-битное представление вещественных чисел), которое неточно соответствует значению `1.1`:

```
(double (float 1.1))
;= 1.100000023841858
```

То же самое можно наблюдать и в Java, где следующее выражение вернет `false`:

```
1.1f == 1.1d
```

Причины этого уходят корнями в спецификацию IEEE представления вещественных чисел. Некоторые языки, такие как Ruby и Python, решают эту проблему, поддерживая единственное представление вещественных чисел с ограниченной точностью.

Оптимизация производительности операций с числами

Clojure дает все необходимое сырье для построения кратких и выразительных моделей, а также для реализации математических алгоритмов и алгоритмов обработки данных. Однако в некоторых случаях производительность имеет большее значение, чем выразительность – это то, что привлекает программистов, которые предпочли бы, чтобы языки высокого уровня позволяли писать фрагменты программ, особенно чувствительные к производительности, на таких языках, как C и Fortran. Да, это прекрасный способ решения проблем с производительностью и его всегда можно рассматривать, как запасной вариант[1], но он очень часто является источником лишних сложностей на этапе интеграции, сборки и развертывания (не говоря уже о снижении продуктивности разработчика). Таким образом, чем больше возможностей будет давать нам высокоуровневый язык, тем большей продуктивности мы сможем достичь.

Так как же обеспечить максимальную производительность программного кода на Clojure, выполняющего операции с числами?

Используйте простые типы. Как говорилось в разделе «Clojure имеет смешанную модель числовых типов» выше, простые типы не отягощены накладными расходами на размещение в памяти и утилизацию сборщиком мусора, свойственными упакованным числовым типам, и большинство операций с ними реализованы на очень низком уровне (нередко даже на аппаратном). При прочих равных условиях, один и тот же алгоритм[2], реализованный на основе простых типов, часто будет показывать производительность на порядки выше, чем реализации на основе упакованных типов. Соблюдение этого правила позволит вам писать на Clojure реализации алгорит-

---

[1] Вы всегда можете воспользоваться библиотеками на низкоуровневых языках из JVM и Clojure. Например, библиотека https://github.com/Chouser/clojure-jna для Clojure обеспечивает простой и выразительный способ использования библиотеки JNA (https://github.com/twall/jna).

[2] Алгоритм с экспоненциальным падением производительности никогда не будет превосходить по быстродействию алгоритм с линейным падением производительности, независимо от того, использует он простые числовые типы или нет. Это долгий разговор: сначала убедитесь, что выбрали правильный алгоритм, и только потом приступайте к его оптимизации, если это необходимо.

мов, которые по своей производительности не будут уступать аналогичной функциональности, написанной на языке Java.

Избегайте коллекций и последовательностей. Как следствие из правила, предлагающего использовать простые типы везде, где производительность имеет большое значение, старайтесь избегать использования коллекций и последовательностей. Как упоминалось во врезке «Зачем вообще нужны типы Long и Double, если они не дают никаких семантических преимуществ перед родственными простыми типами?» выше, коллекции работают исключительно с объектами и не могут хранить значения простых типов. Когда значение простого типа добавляется в коллекцию, оно автоматически преобразуется в экземпляр соответствующего класса-обертки. Этот подход, примиряющий системы на основе простых и объектных типов, называется *автоматической упаковкой* (autoboxing), и наследуется языком Clojure от JVM.

Например, добавление значения типа `double` в список приведет к созданию экземпляра соответствующего класса-обертки (`java.lang.Double`), инициализированного значением простого типа `double`. Несмотря на то, что размещение объектов в памяти и их утилизация сборщиком мусора выполняются в JVM относительно быстро, программа работает гораздо быстрее, когда эти операции отсутствуют вообще. То есть, если вам действительно важна высокая производительность, избегайте использования коллекций и последовательностей, влекущих за собой лишние накладные расходы.

В случае отказа от использования коллекций и последовательностей, вполне естественно обратить свои взоры на массивы значений простых типов.

Объявление функций, принимающих и возвращающих значения простых типов

Когда Clojure компилирует функции, он генерирует соответствующие классы, реализующие `clojure.lang.IFn` — один из Java-интерфейсов в Clojure. Интерфейс `IFn` определяет ряд методов `invoke`; эти методы вызываются за кулисами, когда вы производите вызов функции[1].

---

[1] И эти же методы используются, когда функция вызывается из Java; дополнительные подробности вызова Clojure-функций из Java (b из других языков на базе JVM) описываются в главе 9.

Все аргументы и возвращаемые значения в пределах функций являются экземплярами класса Object. Все эти методы `invoke` принимают аргументы и возвращают значения базового типа `java.lang.Object`. Это обеспечивает динамическую типизацию в Clojure (то есть, реализации ваших функций определяют диапазон допустимых типов аргументов, а не конкретные статические типы), но вынуждает JVM упаковывать значения простых типов, передаваемых в качестве аргументов или возвращаемых в виде результатов. Поэтому, если вы вызываете Clojure-функцию с простым значением в аргументе, например, со значением типа `long`, этот аргумент будет упакован в объект `Long`, чтобы обеспечить соответствие сигнатуре функции. Аналогично, если функция возвращает значение простого типа, в действительности возвращается значение типа `Object`, что гарантирует получение вызывающим кодом упакованного значения.

Несмотря на то, что Clojure является динамическим языком программирования, он распознает информацию о типах (если она передается) и использует ее для оптимизации производительности. Информация о типах[1] на основе классов позволяют компилятору Clojure избежать необходимости использовать механизм рефлексии при взаимодействиях с Java (таких как вызовы методов класса `Object` и так далее), но эта информация не изменяет сигнатуры методов, реализующих функцию – эти методы `invoke` все так же продолжают принимать значения типа `Object`. С другой стороны, Clojure также поддерживает статические объявления аргументов и возвращаемых значений функций, позволяя использовать простые типы: в частности типы `double` и `long`.

Убедиться в этом можно, исследовав методы классов, генерируемых в результате компиляции функций. Рассмотрим в качестве примера функцию, принимающую единственный аргумент без указания его типа:

```
(defn foo [a] 0)
;= #'user/foo
(seq (.getDeclaredMethods (class foo)))
;= (#<Method public java.lang.Object user$foo.invoke(java.lang.Object)>)
```

Обратите внимание, что в результате объявляется метод `invoke` с единственным аргументом. Как и ожидалось, он принимает один

[1] Подробно об этом рассказывается в разделе «Указание типов для производительности», в главе 9.

аргумент типа `Object` и возвращает значение этого же типа. Мы можем также убедиться, что добавление указания типа на основе класса не оказывает влияния на сигнатуру метода `invoke`:

```
(defn foo [^Double a] 0)
;= #'user/foo
(seq (.getDeclaredMethods (class foo)))
;= (#<Method public java.lang.Object user$foo.invoke(java.lang.Object)>)
```

Даже при том, что семантически `Double` является числом, это все-таки класс из-за чего аргумент получает тип `Object`.

Попробуем снова использовать объявления простых типов, `^long` и `^double`:

```
(defn round ^long [^double a] (Math/round a))
;= #'user/round
(seq (.getDeclaredMethods (class round)))
;= (#<Method public java.lang.Object user$round.invoke(java.lang.Object)>
;=  #<Method public final long user$round.invokePrim(double)>)
```

Как видите, Clojure использовал объявления простых типов и сгенерировал метод, принимающий и возвращающий значение простого типа с именем `invokePrim`, принимающий значение простого типа `double` и возвращающий значение простого типа `long`. Вызов этой функции со значением `double` теперь будет выполняться так же быстро, как если бы метод `invokePrim` был написан на Java.

Однако она не будет принимать несоответствующие аргументы:

```
(round "string")
;= #<ClassCastException java.lang.ClassCastException:
;=   java.lang.String cannot be cast to java.lang.Number>
```

Обратите внимание: в сообщении говорится, что `"string"` не является объектом типа `Number`. Фактически мы можем передать любое упакованное число... при условии, что оно будет находиться в ожидаемом диапазоне:

```
(defn idem ^long [^long x] x)
;= #'user/idem
(idem 18/5)
;= 3
(idem 3.14M)
```

```
;= 3
(idem 1e15)
;= 1000000000000000
(idem 1e150)
;= #<IllegalArgumentException java.lang.IllegalArgumentException:
;=   Value out of range for long: 1.0E150>
```

С другой стороны, можно заметить, что типичный метод `invoke` с аргументом и возвращаемым значением типа `Object` никуда не пропал – он остался для поддержки ситуаций, когда функция будет вызываться с упакованным числовым аргументом. Это означает, что можно продолжать использовать функцию, содержащую объявления простых типов аргументов, с функциями высшего порядка[1], такими как `map` и `apply`:

```
(map round [4.5 6.9 8.2])
;= (5 7 8)
(apply round [4.2])
;= 4
```

И указания на типы, и объявления простых типов могут использоваться в определениях полей (и, соответственно, конструкторов) в формах `deftype` и `defrecord`, которые подробно обсуждались в разделе «Определение собственных типов», в главе 6.

Внимание. Функции, поддерживающие простые типы, не могут принимать более четырех аргументов. Предыдущий код выглядит как подарок: вся выразительная мощь языка Clojure соединилась с эффективностью простых числовых типов JVM. К сожалению, здесь имеется один недостаток: любые функции на языке Clojure, объявляющие, что принимают и возвращают значения простых типов, не могут принимать более четырех аргументов:

```
(defn foo ^long [a b c d e] 0)
;= #<CompilerException java.lang.IllegalArgumentException:
;=   fns taking primitives support only 4 or fewer args>
```

Это обусловлено особенностями реализации: чтобы обеспечить упомянутую выше эффективность без помощи строгой статической компиляции, для каждой возможной сигнатуры функции, принимающей аргументы простых типов, должен быть определен отдельный интерфейс. Даже для

[1] Чтобы вспомнить, что такое функции высшего порядка, обращайтесь к разделу «Функции, как сущности первого порядка, и функции высшего порядка» в главе 2.

трех возможных типов аргументов и возвращаемого значения (`double`, `long` и `Object`) и четырех максимально возможных аргументов потребуется создать несколько сотен различных интерфейсов.

Ошибки и предупреждения, вызванные несоответствием типов

Возможность объявлять простые типы для аргументов и возвращаемых значений может приводить к противоречивым ситуациям. Даже при том, что Clojure – это динамический язык, его компилятор будет генерировать ошибки компиляции при обнаружении проблем либо в результате непосредственного анализа, либо при попытке вывести тип:

```
(defn foo ^long [^int a] 0)
;= #<CompilerException java.lang.IllegalArgumentException:
;=   Only long and double primitives are supported>
(defn foo ^long [^double a] a)
;= #<CompilerException java.lang.IllegalArgumentException:
;=   Mismatched primitive return, expected: long, had: double>
```

Аналогично, если переменную `*warn-on-reflection*` связать со значением `true`, компилятор будет выводить предупреждения при обнаружении попытки выполнить форму `recur` со значениями, требующими упаковки, потому что их типы не будут соответствовать объявленному (или выведенному) типу привязки. Рассмотрим простой цикл, выполняющий отсчет в обратном порядке от 5 до 0:

```
(set! *warn-on-reflection* true)
;= true
(loop [x 5]
  (when-not (zero? x)
    (recur (dec x))))
;= nil
```

Опираясь на указанный литерал, механизм вывода типов определит, что `x` имеет тип `long`, и тип аргумента формы `recur` (результат выражения `(dec x)`) так же будет определен как `long`, поэтому здесь не возникает никакого противоречия. Однако, если использовать оператор `dec'`, автоматически расширяющий тип, может возникнуть ситуация, когда аргумент формы `recur` будет иметь тип `BigInt`; Clojure обнаружит это:

```
(loop [x 5]
  (when-not (zero? x)
    (recur (dec' x))))
; NO_SOURCE_FILE:2 recur arg for primitive local:
;                x is not matching primitive, had: Object, needed: long
; Auto-boxing loop arg: x
;= nil
```

То же самое произойдет, если при наличии привязки типа `long` попытаться передать форме `recur` аргумент несовместимого простого типа, такого как `double`:

```
(loop [x 5]
  (when-not (zero? x)
    (recur 0.0)))
; NO_SOURCE_FILE:2 recur arg for primitive local:
;                x is not matching primitive, had: double, needed: long
; Auto-boxing loop arg: x
;= nil
```

Такого рода предупреждения появляются не только в результате проверки простых типов в теле функции. Ниже представлена функция, возвращающая значение типа `double`, что провоцирует появление предупреждения при попытке вызвать `recur`, возвращающую значение для привязки типа `long`:

```
(defn dfoo ^double [^double a] a)
;= #'user/dfoo
(loop [x 5]
  (when-not (zero? x)
    (recur (dfoo (dec x)))))
; NO_SOURCE_FILE:2 recur arg for primitive local:
;                x is not matching primitive, had: double, needed: long
; Auto-boxing loop arg: x
;= nil
```

В таких случаях можно избежать упаковки значений и появления предупреждений, используя `long`, одну из функций приведения к простым типам, гарантируя тем самым, что будет использоваться простое значение нужного типа:

```
(loop [x 5]
  (when-not (zero? x)
    (recur (long (dfoo (dec x))))))
;= nil
```

Функции приведения типов в языке Clojure – `short`, `int`, `long`, `double`, `float` и `boolean` – полезны, только когда необходимо избавиться от предупреждений компилятора, связанных с использованием механизма рефлексии и автоматической упаковки типов. Например, использование функции `long` в примере выше устраняет обращение к механизму автоматической упаковки типов; здесь мы использовали значение типа `double` чтобы избавиться от вызова механизма рефлексии:

```
(defn round [v]
  (Math/round v))
; Reflection warning, NO_SOURCE_PATH:2 - call to round can't be resolved.
;= #'user/round
(defn round [v]
  (Math/round (double v)))
;= #'user/round
```

В частности, при использовании за пределами таких контекстов, функции приведения типов не возвращают значения типов, которые они обозначают:

```
(class (int 5))
;= java.lang.Long
```

При таком подходе они больше напоминают указания типов (обсуждавшиеся в разделе «Указание типов для производительности», в главе 9) и должны использоваться соответственно.

Используйте простые массивы осмысленно

В разделе «Массивы», в главе 9, уже рассказывалось, как можно использовать массивы Java из Clojure, однако имеется несколько важных аспектов, которые необходимо учитывать при работе с массивами значений простых типов.

Изолированное изменение локальных массивов вполне допустимо. Согласно известной поговорке про звук падающего дерева в лесу, который некому услышать, бывают ситуации, когда использование изменяемых массивов не только допустимо, но разумно и не противоречит идиомам языка Clojure.

Несмотря на то, что Clojure настойчиво подталкивает нас к применению функционального стиля программирования – включая ис-

пользование неизменяемых структур данных и абстракции последовательностей, разделение понятий состояния и идентичности, и все остальное, о чем рассказывалось в первой части книги – это в первую очередь практичный язык. Он не чинит никаких препятствий тем, кому действительно требуется использовать изменяемые массивы, чтобы добиться желаемой производительности[1]. Более того, пока изменяемые массивы используются изолированно (то есть, они не выходят за пределы кода, где они действительно необходимы) и локально (то есть, изменяемые массивы не являются глобальными или аргументами ваших функций), можно смело считать, что ваш код все еще имеет функциональную природу, потому что сохраняет семантику идемпотентности.

Типичным примером оправданного использования изменяемых массивов является создание гистограмм. В примере 9.20 мы уже использовали функцию `frequencies` с целью создания гистограмм для различных наборов данных, а теперь посмотрим, как можно реализовать свою аналогичную функцию. Допустим, что нам предстоит анализировать данные простого целочисленного типа. Для их представления наиболее подходящей структурой мог бы быть вектор:

```
(defn vector-histogram
  [data]
  (reduce (fn [hist v]
            (update-in hist [v] inc))
    (vec (repeat 10 0))
    data))
```

Посмотрим, какую производительность можно получить с его применением:

```
(def data (doall (repeatedly 1e6 #(rand-int 10))))    ❶
;= #'user/data
(time (vector-histogram data))
; "Elapsed time: 505.409 msecs"
;= [100383 100099 99120 100694 100003 99940 100247 99731 99681 100102]
```

[1] Это та же самая философия, что оправдывает применение переходных структур данных, описанных в разделе «Переходные структуры данных», в главе 3, и затрагивалась в разделе «Типы», в главе 6, где говорилось о возможности определения изменяемых полей в форме `deftype`.

❶ Наш набор данных – простая последовательность случайных чисел Long (функция doall гарантирует, что она будет реализована (realized) полностью).

Функция vector-histogram использует неизменяемый вектор – весьма оптимизированную структуру данных, но недостаточно эффективную в данном случае[1]. Попробуем использовать массив значений простого типа long:

Пример 11.1. Создание гистограммы с помощью временного массива

```
(defn array-histogram
  [data]
  (vec
    (reduce (fn [^longs hist v]
              (aset hist v (inc (aget hist v)))
              hist)
            (long-array 10)
            data)))
```

Эта версия работает *намного* быстрее, примерно в 20 раз:

```
(time (array-histogram data))
; "Elapsed time: 25.925 msecs"
;= [100383 100099 99120 100694 100003 99940 100247 99731 99681 100102]
```

Массив здесь как нельзя к месту:

1. Его применение соответствует используемой модели и не накладывает ограничений на типы исходных данных (функция reduce способна обрабатывать любые упорядоченные коллекции, включая массивы).
2. Изменяемый массив используется локально, он не покидает границ функции. То есть, array-histogram является чистой функцией, и отлично будет уживаться с другими такими же функциями.
3. Использование массива не отразилось на семантике, ожидаемой пользователем, включая возврат той же самой конкретной структуры данных (вектора), что возвращает vector-histogram.

[1] Применение переходного вектора позволит получить существенный прирост производительности, но вдвое ниже, чем применение массива, как будет показано ниже.

Механика массивов значений простых типов

Чтобы создать массив из имеющейся коллекции, можно воспользоваться функцией `into-array` или `to-array`. Последняя всегда возвращает массив объектов, а первая – массив значений, тип которых определяется типом первого элемента в заданной коллекции, или массив указанного супертипа:

```
(into-array ["a" "b" "c"])
;= #<String[] [Ljava.lang.String;@4413515e>
(into-array CharSequence ["a" "b" "c"])        ❶
;= #<CharSequence[] [Ljava.lang.CharSequence;@5acad437>
```

❶ Явно производит массив указанного супертипа на основе коллекции значений, что может пригодиться при взаимодействиях с некоторыми Java API, требующими такие массивы.

Функцию `into-array` можно также использовать для создания массивов значений простых типов из коллекций значений упакованных типов, указывая класс, соответствующий желаемому простому типу:

```
(into-array Long/TYPE (range 5))
;= #<long[] [J@21e3cc77>
```

В Clojure имеется несколько вспомогательных функций для создания массивов значений простых или ссылочных типов: `boolean-array`, `byte-array`, `short-array`, `char-array`, `int-array`, `long-array`, `float-array`, `double-array` и `object-array`. Они принимают единственный аргумент – желаемый размер массива или исходную коллекцию:

```
(long-array 10)
;= #<long[] [J@12ee6d57>
(long-array (range 10))
;= #<long[] [J@676982f8>
```

При желании всем этим функциям, кроме `object-array`, можно передать одновременно и размер, и исходную коллекцию. Если размер исходной коллекции окажется меньше указанного значения, в конец массива будут записаны значения по умолчанию соответствующего типа:

```
(seq (long-array 20 (range 10)))
;= (0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0)
```

Функция make-array используется для создания новых пустых массивов произвольного размера или размерностей, инициализированных значениями по умолчанию указанного типа (nil – для объектных типов, false – для значений типа Boolean, и ноль – для значений простых числовых типов):

```
(def arr (make-array String 5 5))
;= #'user/arr
(aget arr 0 0)
;= nil
(def arr (make-array Boolean/TYPE 10))
;= #'user/arr
(aget arr 0)
;= false
```

Классы массивов. Функция make-array позволяет создавать массивы любых размерностей, но иногда бывает необходимо определить класс, соответствующий типу массива. Например, чтобы распространить протокол на массив определенного типа, как показано в примере 6.2. Получить класс массива всегда можно с помощью функции class:

```
(class (make-array Character/TYPE 0 0 0))
;= [[[C
```

Но было бы неправильно создавать массив, только чтобы получить экземпляр класса Class, это все равно, что создавать массив некоторой длины, чтобы получить ссылку на число, определяющее эту длину.

Обратили внимание на необычное представление класса массива в выводе предыдущего примера? Именно такие имена даются в JVM классам, которые соответствуют массивам. Каждый класс-обертка имеет статическое поле TYPE со значением соответствующим простому типу, для вложенных массивов нет соответствующих предопределенных классов. Поэтому для определения классов таких типов можно использовать Class/forName, используя нотацию JVM, например:

```
(Class/forName "[[Z")
;= [[Z
(.getComponentType *1)
;= [Z
(.getComponentType *1)
;= boolean
```

Форма записи выглядит немного таинственно, но она не является чем-то необычным. Каждая открывающая квадратная скобка соответствует одной размерности массива, то есть обозначение `"[Z"` соответствует одномерному массиву значений `boolean`, `"[[Z"` соответствует двумерному массиву значений `boolean`, и так далее. За каждым простым типом зарезервирован свой символ:

- `Z` — `boolean`;
- `B` — `byte`;
- `C` — `char`;
- `J` — `long`;
- `I` — `int`;
- `S` — `short`;
- `D` — `double`;
- `F` — `float`.

Указание типов массивов. Обычно, если Clojure не может определить тип массива, с которым вы работаете, операции доступа и изменения будут выполняться с применением механизма рефлексии, что делает бессмысленным использование массивов для локальной оптимизации. Ниже приводится перечень указаний типов, специально предназначенных для массивов:

```
^objects
^booleans
^bytes
^chars
^longs
^ints
^shorts
^doubles
^floats
```

Порядок их применения можно увидеть в реализации функции `array-histogram`, в примере 11.1. Если опустить указание типа для аргумента `hist` в вызове функции `reduce`, обе операции, `aget` и `aset`, выполнялись бы с привлечением механизма рефлексии, что привело бы *к увеличению времени выполнения в 88 раз*, в сравнении с версией, использующей неизменяемые векторы!

Чтение и изменение. Формы `aget` и `aset` реализуют операции чтения и изменения элементов массива, соответственно:

```
(let [arr (long-array 10)]
(aset arr 0 50)
```

```
(aget arr 0))
;= 50
```

Чтобы исключить обращение к механизму рефлексии, обеим операциям следует подсказать тип массива, что легко сделать, использовав синтаксис указания типа. В данном примере известно, что `arr` является массивом значений типа `long`, благодаря объявлению локальной привязки.

Чтение и изменение элементов многомерных массивов имеет свои особенности, о которых рассказывается чуть ниже.

Применение map и reduce к массивам. Функции `map` и `reduce` удобны в использовании, но они работают с коллекциями, как с обобщенными последовательностями, которые могут хранить только объекты. Поэтому при работе с массивами `map` и `reduce` выполняют упаковку значений простых типов.

Все, что можно сделать в этом случае, – развернуть применение `map` и `reduce` в выражение `loop`, обеспечивающее полную поддержку простых типов. Однако использование формы `loop` чревато ошибками, потому что вынуждает следить за индексами в обрабатываемом массиве.

Чтобы избавить вас от такого бремени, Clojure предоставляет макросы `amap` и `areduce`, моделирующие поведение родственных им функций, но предназначенные специально, чтобы избежать автоматической упаковки при работе с массивами:

```
(let [a (int-array (range 10))]
  (amap a i res
    (inc (aget a i))))
;= #<int[] [I@eaf261a>
(seq *1)
;= (1 2 3 4 5 6 7 8 9 10)
```

Функция `amap` принимает четыре аргумента: «исходный» массив, к элементам которого должно применяться выражение: имя для индекса (здесь `i`), имя для массива результата, который инициализируется как копия исходного массива (здесь `res`), и выражение, результат которого должен записываться в массив результата `res`, в элемент с индексом `i`.

Функция `areduce` работает аналогично:

```
(let [a (int-array (range 10))]
  (areduce a i sum 0
```

```
    (+ sum (aget a i))))
;= 45
```

Здесь `a` и `i` играют ту же роль, что и в примере использования функции `amap`. `sum` — это имя аккумулятора (соответствует первому аргументу функции, передаваемой в вызов `reduce`). За именем аккумулятора следует начальное значение и затем выражение, значение которого станет значением аккумулятора в следующей итерации или результатом формы `areduce` после завершения свертки.

Особенности работы с многомерными массивами. Несмотря на простоту использования `aset` и `aget` при работе с одномерными массивами, при применении их к многомерным массивам следует учитывать некоторые особенности. Хотя «конечные» значения в многомерных массивах являются простыми типами, однако промежуточные уровни занимают объекты массивов (других массивов). Кроме того, поскольку функции `aget` и `aset` не поддерживают многомерные массивы непосредственно, они выполняют операции чтения и записи в многомерные массивы рекурсивно, применяя функцию `apply` для каждого уровня вложенности многомерного массива.

Все это приводит к снижению производительности даже при простой обработке многомерных массивов:

```
(def arr (make-array Double/TYPE 1000 1000))
;= #'user/arr
(time (dotimes [i 1000]
          (dotimes [j 1000]
            (aset arr i j 1.0)
            (aget arr i j))))
; "Elapsed time: 50802.798 msecs"
```

Так как `aset` не поддерживает непосредственно N-мерные массивы, значение `1.0` будет упаковано вызовом функции `apply`. Более того, у нас нет возможности указать или объявить, что `arr` является массивом массивов значений простых типов. Поэтому все операции, вовлеченные в обработку будут выполняться рекурсивно. Единственный способ ускорить `aget` и `aset` — выполнять операции с одномерными массивами, указав их тип.

Чтобы «исправить» проблему, достаточно распаковать многомерный массив вручную, добавив необходимое указание типа:

```
(time (dotimes [i 1000]
          (dotimes [j 1000]
```

```
            (let [^doubles darr (aget ^objects arr i)]
              (aset darr j 1.0)
              (aget darr j)))))
; "Elapsed time: 21.543 msecs"
;= nil
```

Вы не ошиблись, новая версия выполняется *в 2600 раз быстрее* предыдущей наивной версии, применяющей `aset` и `aget` к многомерным массивам, и работает так же быстро, как эквивалентный код на Java[1].

Автоматизация указания типов в операциях с многомерными массивами

Итак, мы узнали, как добиться оптимальной производительности, но описанный подход требует добавлять указания типов и писать дополнительный код, что чревато ошибками: для погружения на каждый новый уровень в многомерном массиве необходимо добавлять отдельную `let`-привязку. Здесь естественно возникает желание написать пару макросов для автоматизации распаковки и добавления указания типа в операциях с многомерными массивами[2]:

Пример 11.2. deep-aget

```
(defmacro deep-aget
  "Возвращает значение элемента многомерного массива подобно функции `aget`,
   но автоматически применяет соответствующее указание типа на каждом шаге
   погружения в массив, согласно указанию типа, добавленному к исходному
   массиву.

   например, (deep-aget ^doubles arr i j)"

  ([array idx]
    `(aget ~array ~idx))                                          ❶
  ([array idx & idxs]
    (let [a-sym (gensym "a")]
```

[1] Эквивалентный код на Java фактически действует точно также, но синтаксис языка Java поддерживает более краткий способ выразить операции с массивами, а его модель статической компиляции упрощает получение вовлеченных в рвботу промежуточных типов.

[2] Это решение первоначально было описано в статье: http://clj-me.cgrand.net/2009/10/15/multidim-arrays.

```
      `(let [~a-sym (aget ~(vary-meta array assoc :tag 'objects) ~idx)] ❷
         (deep-aget ~(with-meta a-sym {:tag (-> array meta :tag)})
                    ~@idxs)))))                                          ❸
```

❶ Если обращение выполняется к одномерному массиву (указан единственный индекс), используется функция `aget` непосредственно и предполагается, что символ массива сопровождается соответствующим указанием типа.

❷ Если последний уровень массива еще не достигнут, следует погрузиться на уровень ниже, добавив к аргументу `array` функции `aget` указание типа `^objects`.

❸ Выполнить рекурсивный вызов `deep-aget` с аргументом `a-sym` (следующий уровень массива), повторно применив указание типа для окончательного уровня массива на случай, если это действительно окажется окончательный уровень. По достижении самого нижнего уровня макрос `deep-aget` будет вызван с единственным индексом ❶ и выполнит заключительный вызов `aget`.

Макрос `deep-aset` использует иной подход, так как ему необходимо использовать `deep-aget` чтобы эффективно выполнить погружение в многомерный массив до уровня, где он наконец сможет применить заключительный вызов `aset`:

Пример 11.3. deep-aset

```
(defmacro deep-aset
  "Записывает значение в многомерный массив подобно функции `aset`,
   но автоматически применяет соответствующее указание типа на каждом шаге
   погружения в массив, согласно указанию типа, добавленному к целевому
   массиву.

   например, (deep-aset ^doubles arr i j 1.0)"

  [array & idxsv]
  (let [hints '{booleans boolean, bytes byte      ❶
                chars char, longs long
                ints int, shorts short
                doubles double, floats float}
        hint (-> array meta :tag)
        [v idx & sxdi] (reverse idxsv)
        idxs (reverse sxdi)
        v (if-let [h (hints hint)] (list h v) v)
        nested-array (if (seq idxs)
                       `(deep-aget ~(vary-meta array assoc :tag 'objects)
                                   ~@idxs)
```

```
                    array)
        a-sym (gensym "a")]
    `(let [~a-sym ~nested-array]
       (aset ~(with-meta a-sym {:tag hint}) ~idx ~v))))
```

❶ Связь между символами, указывающими тип массива, и функциями приведения к простым типам. Если массив `array` снабжен указанием типа, одним из перечисленных здесь, тогда в паре с `aset` и значением, переданным макросу `deep-aset`, будет использоваться соответствующее выражение приведения типа. Это даст пользователю возможность, например, записать в многомерный массив значений типа `double` значение типа `long`, при этом значение `long` будет приведено к типу `double` в вызове `aset` через `(double v)`.

Макросы `deep-aget` и `deep-aset` позволяют добиться такой же производительности, как при распаковке N-мерного массива и использовании указания типа вручную:

```
(time (dotimes [i 1000]
        (dotimes [j 1000]
          (deep-aset ^doubles arr i j 1.0)
          (deep-aget ^doubles arr i j))))
; "Elapsed time: 25.033 msecs"
```

Внимание. Когда переменная `*warn-on-reflection*` установлена в значение `true`, применение `aget` и `aset` к массивам, чей тип не может быть определен, вызывает вывод предупреждения... кроме случаев применения к многомерным массивам!

Визуализация множества Мандельброта в Clojure

Рассмотрим нечто более интересное, чем набившие оскомину примеры измерения производительности генераторов простых чисел и чисел Фибоначчи. Практика визуализации множества Мандельброта (Mandelbrot)[1] (или, фактически, любых фракталов) насчитывает

[1] Введение в множество Мандельброта и математический аппарат, стоящий за ним, а так же описание как его визуализировать можно найти на странице http://en.wikipedia.org/wiki/Mandlebrot_set (http://ru.wikipedia.org/wiki/Множество_Мандельброта – *прим. перев.*). С нашей стороны было

уже много лет, и она послужит отличной демонстрацией оптимизации численных алгоритмов, реализованных на языке Clojure.

Множество Мандельброта определяется комплексным полиномом, применяемым итеративно:

$$z_{k+1} = z_k^2 + c,$$

где c (комплексное число) является членом множества Мандельброта, если z_{k+1} не растет до бесконечности с ростом k, при условии, что z_0 инициализируется нулем. Если для данного значения c последовательность не сходится, говорят, что она *убегает в бесконечность*.

Для начала рассмотрим наивную реализацию множества Мандельброта на языке Clojure[1], включающую пару вспомогательных функций для отображения результатов на экране:

Пример 11.4. Множество Мандельброта на Clojure

```
(ns clojureprogramming.mandelbrot
  (:import java.awt.image.BufferedImage
           (java.awt Color RenderingHints)))
(defn- escape
  "Возвращает целое число, определяющее максимальное число итераций, которые
   должны быть выполнены, прежде чем можно будет сказать убегает ли значение
   z в бесконечность (на основе компонентов `a` и `b`). Если z не убегает,
   возвращается -1."
  [a0 b0 depth]
  (loop [a a0
         b b0
         iteration 0]
    (cond
      (< 4 (+ (* a a) (* b b))) iteration
```

бы небрежностью не упомянуть фантастическую песню Джонатана Култона (Jonathan Coulton) и видеоклип о множестве Мадельброта и его создателе/первооткрывателе Бенуа Мандельброте (Benoît Mandelbrot): http://www.youtube.com/watch?v=ES-yKOYaXq0.

[1] Возможны еще более простые реализации. Например, с помощью функции `iterate` можно реализовать отложенные вычисления значений комплексного полинома и извлекать столько результатов из головы этой «ленивой» последовательности, сколько требует максимальное значение счетчика итераций. Такие реализации получаются более короткими, но из-за использования «ленивых» последовательностей и коллекций, результаты вычислений будут упаковываться, что будет существенно замедлять работу алгоритма.

```
       (>= iteration depth) -1
       :else (recur (+ a0 (- (* a a) (* b b)))
                    (+ b0 (* 2 (* a b)))
                    (inc iteration)))))

(defn mandelbrot
  "За определенное число итераций выясняет принадлежность к множеству
   Мандельброта области, определяемой `rmin`, `rmax`, `imin` и `imax`
   (действительные и мнимые компоненты z, соответственно).

   В число необязательных именованных аргументов входят
   `:depth` (максимальное число итераций для определения убегания точки
   в бесконечность), `:height` (высота 'пикселя' при отображении)
   и `:width` (ширина 'пикселя' при отображении).

   Возвращает последовательность векторов, строк матрицы, содержащих
   количество итераций, при котором было обнаружено убегание соответствующей
   точки в бесконечность. Значение -1 соответствует точкам, для которых не
   было выявлено убегание за заданное число итераций `depth`, то есть
   точкам, принадлежащим множеству. Эти числа можно использовать для
   управления визуализацией множества Мандельброта."

  [rmin rmax imin imax & {:keys [width height depth]
                          :or {width 80 height 40 depth 1000}}]
  (let [rmin (double rmin)
        imin (double imin)
        stride-w (/ (- rmax rmin) width)
        stride-h (/ (- imax imin) height)]
    (loop [x 0
           y (dec height)
           escapes []]
      (if (== x width)
        (if (zero? y)
          (partition width escapes)
          (recur 0 (dec y) escapes))
        (recur (inc x) y (conj escapes (escape (+ rmin (* x stride-w))
                                               (+ imin (* y stride-h))
                                               depth)))))))
(defn render-text
  "Выводит в простом текстовом представлении множество Мандельброта,
   полученное вызовом функции `mandelbrot`."
  [mandelbrot-grid]
```

```
  (doseq [row mandelbrot-grid]
    (doseq [escape-iter row]
      (print (if (neg? escape-iter) \* \space)))
    (println)))

(defn render-image
  "Получает матрицу с множеством Мандельброта, возвращаемую функцией
   `mandelbrot`, и возвращает BufferedImage с теми же размерами, что и
   матрица, и с палитрой, включающей оттенки серого цвета."
  [mandelbrot-grid]
  (let [palette (vec (for [c (range 500)]
                       (Color/getHSBColor 0.0 0.0
                                          (/ (Math/log c) (Math/log 500)))))
        height (count mandelbrot-grid)
        width (count (first mandelbrot-grid))
        img (BufferedImage. width height BufferedImage/TYPE_INT_RGB)
        ^java.awt.Graphics2D g (.getGraphics img)]
    (doseq [[y row] (map-indexed vector mandelbrot-grid)
            [x escape-iter] (map-indexed vector row)]
      (.setColor g (if (neg? escape-iter)
                     (palette 0)
                     (palette (mod (dec (count palette)) (inc escape-iter)))))
      (.drawRect g x y 1 1))
    (.dispose g)
    img))
```

Функция `mandelbrot` возвращает матрицу, каждая ячейка которой хранит число итераций, потребовавшееся, что установить, что данная точка убегает в бесконечность; точки, не убегающие в бесконечность и, соответственно, являющиеся членами множества, обозначаются значением –1. Самое простое (и грубое) изображение множества Мандельброта можно получить, выводя звездочки и пробелы в консоли, что и реализует функция `render-text`[1]:

[1] Естественная природа текста – когда высота символов превышает их ширину – требует использовать искаженное отношение сторон (здесь используется отношение 80×40), чтобы результат выглядел как изображение с отношением сторон 1:1.

```
(render-text (mandelbrot -2.25 0.75 -1.5 1.5 :width 80 :height 40 :depth 100))
                                                ***
                                              ******
                                               ****
                                          **    ************
                                           ***********************
                                         **********************
                                        ************************
                                      ***************************
                          *******     ******************************
                         ***********  ******************************
                        ************ ******************************
 ************************************************************
                        ************ ******************************
                         ***********  ******************************
                          *******     ******************************
                                      ***************************
                                        ************************
                                         **********************
                                           ***********************
                                          **    ************
                                               ****
                                              ******
                                                ***
```

Однако, чтобы увеличить изображение (и выявить более тонкие детали), нам потребовалось бы выполнить намного больше итераций функции `escape`. Хронометраж создания изображения размером 1600×1200 показывает, что реализация выше не обладает той скоростью, какой хотелось бы:

```
(do (time (mandelbrot -2.25 0.75 -1.5 1.5
              :width 1600 :height 1200 :depth 1000))
nil)
; "Elapsed time: 82714.764 msecs"
```

Если вам потребуется создать приложение для исследования множества Мандельброта в интерактивном режиме (или для исследований в автоматическом режиме), такая производительность окажется в принципе неприемлемой.

Совершенно очевидно, что наиболее жаркой частью в этом коде является цикл `loop` в функции `escape`. Мы не можем упростить вычисления, выполняемые в цикле – их сложность обусловлена

определением полинома множества Мандельброта – но мы можем избавиться от упаковки числовых значений.

Как отмечалось в разделе «Объявление функций, принимающих и возвращающих значения простых типов» выше, функции в Clojure реализуются как Java-методы, принимающие аргументы типа `java.lang.Objects`, поэтому значения `a0` и `b0` простого типа `double`, которые передаются функцией `mandelbrot` в вызов `escape` (действительный и мнимый компоненты `z` в полиноме, описывающем множество Мандельброта), в конечном итоге упаковываются в объекты `Double`. Это каскадом распространяется на типы привязок в форме `loop` и сводит на нет производительность функции: инициализация `a` и `b` значениями `a0` и `b0` автоматически приводит их к типу `Double`, поэтому все арифметические операции в функции `escape` начинаются с распаковки исходных значений и заканчиваются обязательной упаковкой результатов, когда они связываются с символами `a` и `b` в форме `recur`.

Решение заключается в первую очередь в устранении нетипизированных привязок; обратите внимание на объявления типов `^double` аргументов `a0` и `b0` в улучшенной версии `escape`, где все остальное осталось без изменений.

Пример 11.5. Улучшенная функция escape с объявлениями типов аргументов

```
(defn- escape
  [^double a0 ^double b0 depth]
  (loop [a a0
         b b0
         iteration 0]
    (cond
      (< 4 (+ (* a a) (* b b))) iteration
      (>= iteration depth) -1
      :else (recur (+ a0 (- (* a a) (* b b)))
                   (+ b0 (* 2 (* a b)))
                   (inc iteration)))))
```

Теперь класс, сгенерированный для функции `escape`, будет принимать значения простых типов `double` в первом и втором аргументах. Все остальное сделает механизм определения типов в языке Clojure: привязки `a` и `b` в форме `loop` получат тип `double`, и все арифметические операции будут выполняться исключительно со значениями простых типов, минуя упаковку результатов при передаче аргумен-

тов форме recur и при повторном связывании с соответствующими именами в форме loop.

И какой же результат мы получили? Производительность выросла на порядок:

```
(do (time (mandelbrot -2.25 0.75 -1.5 1.5
            :width 1600 :height 1200 :depth 1000))
  nil)
; "Elapsed time: 8663.841 msecs"
```

Добившись такой производительности, можно двинуться вперед и заняться исследованием множества с помощью функции render-image, возвращающую растровое представление матрицы с членами множества Мандельброта, как показано на рис. 11.2 и 11.3.

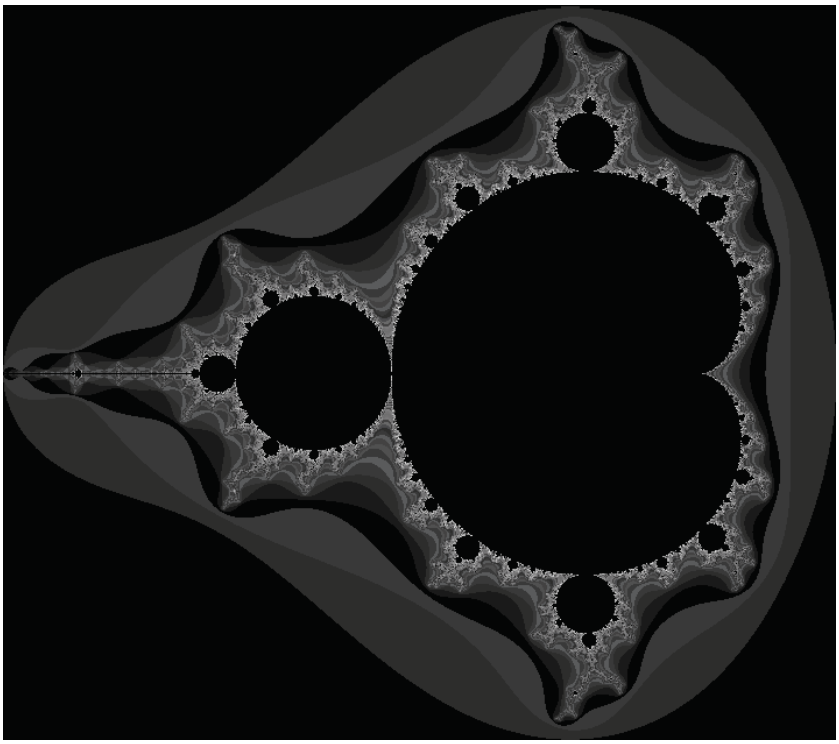

Рис. 11.2. (render-image
(mandelbrot -2.25 0.75 -1.5 1.5 :width 800 :height 800 :depth 500))

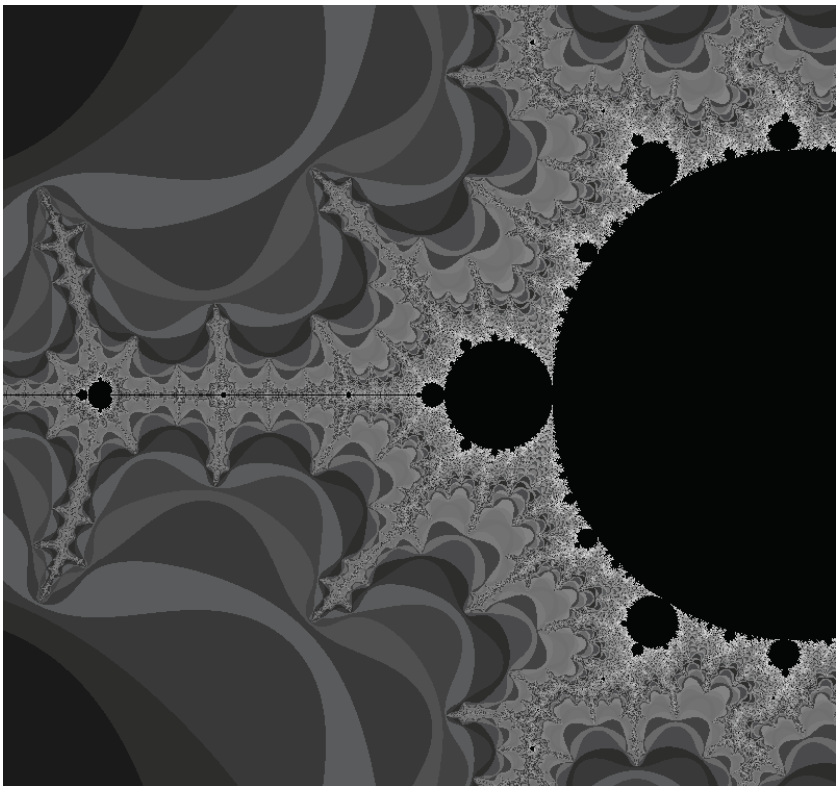

Рис. 11.3. (render-image
(mandelbrot -1.5 -1.3 -0.1 0.1 :width 800 :height 800 :depth 500))

Функция `render-image` использует черно-белую палитру градаций серого цвета. Изменение ее для использования полного набора цветов мы оставляем читателю в качестве самостоятельного упражнения.

Примечание. Если вы используете типичную текстовую оболочку REPL[1], выражения, указанные в подписях к рисункам, вернут экземпляры `Buff-`

[1] Оболочки REPL, поддерживающие вывод графики, отобразят изображение на экране немедленно; то есть вызов `render-image` в таких оболочках REPL сразу же отобразит изображение и вам не придется сохранять его на диск или как-то иначе извлекать данные изображения из оболочки. Например, загляните сюда: http://cemerick.com/2011/10/26/enabling-richer-interactions-in-the-clojure-repl.

eredImage. Изображения можно сохранить на диск с помощью класса ImageIO, входящего в комплект JDK, и его метода write, выполнив команду:

```
(javax.imageio.ImageIO/write *1 "png" (java.io.File. "mandelbrot.png"))
```

которая сохранит изображение, привязанное к переменной *126, в формате PNG в файле mandelbrot.png в текущем каталоге. Достаточно легко самому написать функцию, выполняющую этот вызов.

Существует целое множество алгоритмических усовершенствований, позволяющих ускорить вычисление множества Мандельброта и еще больше увеличить производительность функции mandelbrot, но все они далеко выходят за рамки обсуждаемой темы. Упомянем лишь некоторые из них, касающиеся непосредственно языка Clojure.

- Чтобы не выполнять все вычисления в императивном цикле loop, их можно оформить в виде «ленивой» последовательности, что даст возможность распараллелить вычисления с помощью pmap.
- Возвращая матрицу (последовательность строк с количествами итераций для точек, убегающих в бесконечность) из функции mandelbrot мы получаем немалую гибкость. Как было показано выше, одну и ту же структуру с результатами можно использовать для построения как текстового, так и растрового представления. Аналогично можно было бы создать функцию отображения множества Мандельброта в трехмерном пространстве, ничего не меняя в функциях mandelbrot и escape. Однако создание матрицы влечет за собой определенные накладные расходы, поэтому, возможно, имело бы смысл передавать функции escape (через mandelbrot) ссылку на функцию обратного вызова, чтобы можно было выполнять операции с точками непосредственно (например, при выводе символа в консоль или пикселя в растровое изображение) без распределения памяти для коллекций и последовательностей. Разумеется, эта функция обратного вызова должна принимать аргументы простых типов с соответствующими объявлениями, как это сделано в функции escape, чтобы избежать автоматической упаковки.

Глава 12. Шаблоны проектирования

Шаблоны проектирования – это универсальные решения постоянно возникающих задач. А так как они несут на себе отпечаток объектно-ориентированного программирования, при их описании используется набор терминов, хорошо знакомых программистам на Java и Ruby.

С другой стороны, шаблоны проектирования могут провоцировать на создание слишком подробного и повторяющегося кода. Пол Грэм (Paul Graham) заметил, что наличие и использование шаблонов проектирования в языке может служить признаком его слабости, а не последовательного подхода к решению задач:

> Наличие шаблонов в моих программах служит для меня признаком недостатков. Форма программы должна отражать только решаемую задачу. Любые другие закономерности в коде являются признаком, по крайней мере для меня, что я использую недостаточно мощные абстракции...
>
> – Пол Грэм (Paul Graham),
> http://www.paulgraham.com/icad.html

Грэм был далеко не первым, сделавшим такое наблюдение; немного раньше Питер Норвиг (Peter Norvig) (http://www.norvig.com/design-patterns/) продемонстрировал, что Lisp в частности позволяет упростить или сделать незаметными большинство шаблонов проектирования. Clojure является достойным продолжателем этой традиции: благодаря мощным конструкциям, таким как функции первого порядка (first-class functions), динамической типизации и неизменяемости значений, многие распространенные шаблоны проектирования растворяются как утренний туман. А под видом макросов, Clojure дает инструменты, позволяющие избежать необходимости писать собственный шаблонный код.

Примеры на языке Clojure, относящиеся к категории типичных шаблонов проектирования, разбросаны повсюду в этой книге:

Слушатель (Listener), Наблюдатель (Observer). Реализации этого шаблона, благодаря наличию функций первого порядка и под-

держке динамической типизации, являются обычными функциями, которые вызываются по событиям. Этот шаблон проектирования можно увидеть в примерах функций-наблюдателей для ссылочных типов в разделе «Функции-наблюдатели» главы 4. Если исключить ссылочные типы, при предпочтительном использовании неизменяемых значений область, где могло бы пригодиться слежение за изменяемыми объектами, оказывается слишком узкой.

Абстрактная фабрика (Abstract Factory), Стратегия (Strategy), Команда (Command). Когда имеется несколько реализаций некоторой функциональности – будь то создание значений разных видов и форм или реализация различных вариантов алгоритма – необязательно создавать `FactoryFactory`[1] (фабрику фабрик) или определять контекст вызова нужной реализации алгоритма. В обоих случаях достаточно определить еще одну функцию.

Итератор (Iterator). Потребность в итераторах с лихвой удовлетворяется последовательностями, описанными в разделе «Последовательности», в главе 3, обеспечивающими простое и декларативное их использование с помощью таких функций как `map`.

Адаптер (Adapter), Обертка (Wrapper), Делегат (Delegate). Необходимые в других языках из-за отсутствия гибкости в иерархиях классов, они становятся ненужными, благодаря протоколам, позволяющим определять новые функциональные возможности для существующих типов без обращения к механизмам наследования, делегирования или обертывания. См. главу 6.

Хранитель (Memento). Наслаивание обобщенных API поверх операций с изменяемыми объектами решает многие механические проблемы, но ничего не дает для ликвидации вопиющей сложности обращения с самим изменяемым состоянием. Неизменяемые коллекции и записи позволяют опрокинуть эту стратегию: изменяемые состояния отделяются от значений, сохранение прежних версий этих значений становится простой и недорогой операцией, а смена состояний возлагается на ссылочные типы (каждый из которых поддерживает свой API, наиболее подходящий для реализации той или иной семантики изменения и конкуренции).

Шаблонный метод (Template Method). Ограничения механизма наследования классов, который все еще широко используется и во многих языках часто является единственным способом определе-

---

[1] Оставим эти проблемы на долю экспертов по созданию фабричных функций: http://discuss.joelonsoftware.com/default.asp?joel.3.219431.12.

ния функциональных возможностей, хорошо известны и ежедневно ощущаются миллионами программистов. Этому сомнительному решению проблем лучше придать форму функции высшего порядка, принимающей другие функции, которые реализуют вариации поведения, и одновременно поддерживающей каноническую реализацию общей функциональности. Например, если в программе потребуется иметь возможность вызывать эквивалентные HTTP API различных провайдеров, опираясь на некоторую внутреннюю структуру данных, можно написать функцию высшего порядка, позволяющую определять функциональность для каждого конкретного провайдера в виде отдельных функций, и реализовать в ней операции, общие для всех провайдеров:

```
(defn- update-status*
  [service-name service-endpoint-url request-data-fn]
  (fn [new-status]
    (log (format "Updating status @ %s to %s" service-name new-status))
    (let [http-request-data (request-data-fn new-status)
          connection (-> service-endpoint-url java.net.URL. .openConnection)]
      ;; ...установить метод запроса, параметры, тело в `connection`
      ;; ...выполнить фактический запрос
      ;; ...вернуть результат на основе кода состояния HTTP-ответа
      )))

(def update-facebook-status (update-status* "Facebook" "http://facebook.com/apis/..."
                              (fn [status]
                                {:params {:_a "update_status"
                                          :_t status}
                                 :method "GET"})))

(def update-twitter-status ...)
(def update-google-status ...)
```

Рассмотрим поближе некоторые другие распространенные шаблоны проектирования, ставшие настолько обычной практикой программирования в других языках, что стоит внимательно разобрать их.

Внедрение зависимостей

Во многих объектно-ориентированных языках прием *внедрения зависимостей* дает возможность разорвать связь класса с другими объектами, от которых этот класс зависит. При использовании этого

приема объекты не занимаются инициализацией других объектов, а принимают их в виде параметров, часто волшебным образом инициализированных средой выполнения или контейнером приложений, в котором выполняется программа.

В статических языках, таких как Java, эта возможность достигается с помощью интерфейсов, а не конкретных классов. Взгляните на следующую реализацию идеи зоомагазина:

```
interface IDog {
     public String bark();
}

class Chihuahua implements IDog {
     public String bark() {
          return "Yip!";
     }
}

class Mastiff implements IDog {
     public String bark() {
          return "Woof!";
     }
}

class PetStore {
     private IDog dog;
     public PetStore() {
          this.dog = new Mastiff();
     }

     public IDog getDog() {
          return dog;
     }
}

static class MyApp {
     public static void main(String[] args) {
          PetStore store = new PetStore();
          System.out.println(store.getDog().bark());
     }
}
```

Наш зоомагазин продает только догов. Если потребуется реализовать поддержку других пород собак, нам придется изменить и пере-

компилировать класс PetStore. Чтобы сделать класс PetStore более независимым, его можно переписать, как показано ниже:

```
class PetStore {
     private IDog dog;
     public PetStore(IDog dog) {
          this.dog = dog;
     }

     public IDog getDog() {
          return dog;
     }
}

class MyApp {
     public static void main(String[] args) {
          PetStore store = new PetStore(new Chihuahua());
          System.out.println(store.getDog().bark());
     }
}
```

Теперь порода собак определяется параметром. Конкретная порода «внедряется» в объект класса PetStore через его конструктор. Теперь PetStore не нужно перекомпилировать всякий раз, когда потребуется выбрать другую породу. Более того, PetStore можно использовать с классами, которые пока даже не написаны; достаточно лишь реализовать в этих новых классах интерфейс IDog. Такой подход позволяет легко создавать фиктивные породы собак, например, для тестирования.

Внедрение зависимостей обычно выполняется «контейнером», использующим настройки среды выполнения для автоматической инициализации ключевых объектов с реализациями интерфейсов, которые могут автоматически обнаруживаться в пути поиска classpath или определяться непосредственно в конфигурации. В зависимости от особенностей реализации контейнера, конфигурация может осуществляться отдельным кодом или храниться в XML-файлах.

Clojure выворачивает эту проблему наизнанку. Если в Java bark — это методы классов, реализующих интерфейс IDog, то в Clojure bark можно определить как метод протокола, независимый от конкретных типов.

```
(defprotocol Bark
  (bark [this]))

(defrecord Chihuahua []
  Bark
  (bark [this] "Yip!"))

(defrecord Mastiff []
  Bark
  (bark [this] "Woof!"))
```

Теперь зоомагазин можно реализовать так:

```
(defrecord PetStore [dog])

(defn main
  [dog]
  (let [store (PetStore. dog)]
    (println (bark (:dog store)))))

(main (Chihuahua.))
;= Yip!

(main (Mastiff.))
;= Woof!
```

Да, это все! Определение `PetStore` уместилось в одной короткой строке.

В Java `PetStore` может использовать только объекты, реализующие интерфейс `IDog`. Реализация `PetStore` на Clojure не имеет подобных ограничений: она принимает параметры любых типов. Если для этих типов имеется реализация протокола `Bark`, все будет работать без сучка и задоринки. Данное решение позволяет творить такое, чего язык Java не позволил бы никогда, например, можно распространить протокол `Bark` на классы, созданные сторонними разработчиками. С помощью следующего простого кода:

```
(extend-protocol Bark
  java.util.Map
  (bark [this]
    (or (:bark this)
        (get this "bark"))))
```

...мы можем использовать в качестве собак любые объекты Map, включая Clojure IPersistentMap или HashMap в Java, где строка, описывающая лай (bark) может быть связана с :bark или "bark"[1]:

```
(main (doto (java.util.HashMap.)
          (.put "bark" "Ouah!")))
;= Ouah!

(main {:bark "Wan-wan!"})
;= Wan wan!
```

В записях все поля являются динамическими (если они не объявлены полями простых типов), поэтому здесь не требуется определять типы для управления конфигурацией, как в контейнерах, реализующих внедрение зависимостей (хорошо это или плохо – решать вам). Отсутствие контейнера[2] *расширяет* возможности определения конфигурации. Например, ниже представлен своеобразный «конфигурационный файл» (в действительности – простой файл с данными, которые могут быть прочитаны механизмом чтения Clojure, которые только читаются и не выполняются):

Пример 12.1. petstore-config.clj

```
{:dog #user.Chihuahua{:weight 12, :price "$84.50"}}
```

В разделе «Читаемое представление» (глава 6) говорилось, что форма записи #user.Chihuahua{...} является читаемым представлением именованной записи. Мы полагаем, что она *ничуть не хуже* описаний компонентов (beans) в конфигурационных XML-файлах для Spring, аннотаций @Bean или аналогичных конструкций, предлагаемых другими контейнерами с поддержкой внедрения зависимостей.

Улучшенная фабричная функция PetStore могла бы читать настройки для конкретного окружения:

---

[1] Как изображается лай собак в разных языках можно узнать по адресу: https://en.wikipedia.org/wiki/Bark_%28utterance%29#Representation.

[2] Обратите внимание, что для работы с типами и записями Clojure можно использовать «унаследованные» контейнеры с поддержкой внедрения зависимостей (такие как Spring, Guice и прочие).

Пример 12.2. configured-petstore

```
(defn configured-petstore
  []
  (-> "petstore-config.clj"
    slurp
    read-string
    map->PetStore))
```

Теперь мы всегда будем иметь правильно настроенную запись PetStore в «внедренными» зависимостями:

```
(configured-petstore)
;= #user.PetStore{:dog #user.Chihuahua{:weight 12, :price "$84.50"}}
```

Шаблон Стратегия (Strategy)

Другим распространенным шаблоном проектирования является шаблон Стратегия (Strategy). Этот шаблон позволяет динамически выбирать метод или алгоритм. Допустим, алгоритм сортировки должен выбираться во время выполнения:

```
interface ISorter {
     public sort (int[] numbers);
}

class QuickSort implements ISorter {
     public sort (int[] numbers) { ... }
}

class MergeSort implements ISorter {
     public sort (int[] numbers) { ... }
}

class Sorter {
     private ISorter sorter;
     public Sorter (ISorter sorter) {
          this.sorter = sorter;
     }

     public execute (int[] numbers) {
          sorter.sort(numbers);
     }
```

```
}

class App {
     public ISorter chooseSorter () {
          if (...) {
               return new QuickSort();
          } else {
               return new MergeSort();
          }
     }

     public static void main(String[] args) {
          int[] numbers = {5,1,4,2,3};

          Sorter s = new Sorter(chooseSorter());

          s.execute(numbers);

          //... использовать отсортированный массив numbers
     }
}
```

В этой ситуации Clojure имеет очень простое преимущество перед Java. В Java методы должны помещаться в классы, а в Clojure функции являются сущностями первого порядка. Буквальный перевод примера выше на язык Clojure мог бы выглядеть так:

```
(defn quicksort [numbers] ...)

(defn mergesort [numbers] ...)

(defn choose-sorter
  []
  (if ...
    quicksort
    mergesort))

(defn main
  []
  (let [numbers [...]]
    ((choose-sorter) numbers)))
```

Здесь нет никаких классов. Каждая функция, реализующая семантику алгоритма, может вызываться непосредственно, избавляя нас от необходимости определять классы для реализации алгоритмов.

Мы можем даже не давать имена нашим алгоритмам сортировки – с этой работой прекрасно справятся анонимные функции. Например, ниже представлена анонимная композиции из встроенных функций `sort` и `reverse`, позволяющая выполнять сортировку в обратном порядке:

```
((comp reverse sort) [2 1 3])
;= (3 2 1)
```

Цепочка обязанностей (Chain of Responsibility)

Возможности языка Clojure делают многие шаблоны проектирования ненужными или неуместными, однако есть и такие, которые остаются актуальными и продолжают влиять на архитектуру и реализацию программ. Одним из них является шаблон организации потока управления, называемый «цепочка обязанностей» (chain of responsibility). Этот шаблон предусматривает передачу событий множеству обработчиков. Каждый обработчик может обслужить событие или передать его дальше, другому обработчику. Обработчики объединяются в цепочки. Событие следует по этой цепочке, пока один из обработчиков не определит, что оно должно прекратить дальнейшее распространение.

Этот шаблон позволяет объединять и комбинировать различные обработчики, определенные в разных частях приложения. Ни в одном из обработчиков не требуется знать об этапах процесса обработки, кроме способа передачи события следующему обработчику в цепочке.

Воплощение идеи создания цепочек можно найти во многих областях. Примером могут служить каналы Unix, где текстовые данные передаются от процесса к процессу. Другой пример – Java-сервлеты, где веб-запросы передаются через последовательность фильтров до тех пор, пока не будет сгенерирован ответ.

В Java шаблон цепочка обязанностей можно реализовать, определив серию объектов-процессоров и инициализировать каждый из них указателем на следующий объект-процессор в цепочке.

```
abstract class Processor {
     protected Processor next;
     public addToChain(Processor p) {
          next = p;
```

```
     }

     public runChain(data) {
          Boolean continue = this.process(data);
          if(continue and next != null) {
               next.runChain(data);
          }
     }
     abstract public boolean process(String data);
}

class FooProcessor extends Processor {
     public boolean process(String data) {
          System.out.println("FOO says pass...");
          return true;
     }
}

class BarProcessor extends Processor {
     public boolean process(String data) {
          System.out.println("BAR " + data + " and let's stop here");
          return false;
     }
}

class BazProcessor extends Processor {
     public boolean process(String data) {
          System.out.println("BAZ?");
          return true;
     }
}

Processor chain = new FooProcessor().addToChain(
     new BarProcessor).addToChain(new BazProcessor);

chain.run("data123");
```

В этом примере используется простое соглашение о возвращении логического значения `true`, чтобы показать, что данные не были обработаны и данные должны быть переданы дальше по цепочке, и `false` – чтобы прекратить обработку данных.

В Clojure элементарной единицей выполнения является функция, а не класс или метод. То есть, мы можем организовать цепочку, используя простой прием композиции функций.

```
(defn foo [data]
  (println "FOO passes")
  true)

(defn bar [data]
  (println "BAR" data "and let's stop here")
  false)

(defn baz [data]
  (println "BAZ?")
  true)

(defn wrap [f1 f2]
  (fn [data]
    (when (f1 data)
      (f2 data))))

(def chain (reduce wrap [foo bar baz]))
```

Здесь мы определили функцию `wrap`, принимающую две функции и объединяющую их подобно тому, как это делалось в примере на Java: первой вызывается `f1` и если она возвращает `true`, вызывается `f2`. Таким способом можно составлять цепочки из функций, которые будут вызываться по очереди, пока какая-нибудь не вернет `false` или не будет достигнут конец цепочки. Далее выполняется построение цепочки с помощью простой функции `reduce`.

Ring. Более практичным примером реализации цепочек обработчиков на языке Clojure является библиотека Ring, которая подробно будет обсуждаться в главе 16. Ring – это библиотека обработки запросов к веб-серверу. Обработчиком запроса является функция, принимающая объект запроса в форме ассоциативного массива и возвращающая объект ответа, так же в форме ассоциативного массива.

Функции, изменяющие и обрабатывающие эти запросы и ответы, называются *промежуточными* (middleware). Промежуточные функции объединяются в цепочки, почти так же, как показано в примере выше. Представьте, что первоначально у нас имеется простейший обработчик:

```
(defn my-app
  [request]
  {:status 200
```

```
    :headers {"Content-type" "text/html"}
    :body (format "<html><body>You requested: %s</body></html>"
            (:uri request))})
```

Этот обработчик можно «завернуть» в промежуточную функцию, выполняющую некоторые другие операции. Один обработчик может анализировать данные cookie в запросе и добавлять ключ `:cookie` в ассоциативный массив запроса или ответа. Другой – выполнять аналогичные операции с данными `:session`. Подобные промежуточные функции уже предоставляются библиотекой Ring, но в ней отсутствуют промежуточные функции, реализующие запись информации о запросах в файл журнала; определение такой промежуточной функции могло бы выглядеть, как показано ниже:

```
(defn wrap-logger
  [handler]
  (fn [request]
    (println (:uri request))
    (handler request)))
```

В зависимости от назначения обработчика, он может выполнять операции до или после передачи обрабатываемого запроса следующему обработчику в цепочке, или вообще не передавать запрос дальше. Наша простенькая промежуточная функция журналирования просто выводит содержимое элемента `:uri` каждого запроса в `stdout` и затем вызывает следующий обработчик.

Теперь, наряду с промежуточными функциями обработки данных cookies и сеансов из библиотеки Ring, можно использовать и нашу функцию журналирования:

```
(require '[ring.middleware cookies session])

(def my-app (-> my-app
                wrap-cookies
                wrap-session
                wrap-logger))
```

Этот способ композиции очень напоминает функцию `comp`, о которой рассказывалось в разделе «Композиция функций», в главе 2, поскольку в результате получается функция, вызывающая все обработчики в порядке, обратном их следованию в списке аргументов

формы `->`. Так как `wrap-logger` указана в списке последней, возвращаемая ею функция окажется самой внешней в композиции, а `my-app` — самой внутренней.

Аспектно-ориентированное программирование

Аспектно-ориентированное программирование (Aspect-Oriented Programming, AOP) – это методология, позволяющая отделять *сквозные функции* (crosscutting concerns). В объектно-ориентированном коде реализация поведения часто повторяется в нескольких классах или оказывается разбросанной по нескольким методам. Аспектно-ориентированное программирование (АОП) позволяет вынести общую реализацию за скобки и применять ее к классам без использования механизма наследования.

Типичным примером является реализация определения показателей. Часто бывает желательно определять и записывать данные хронометража или другую отладочную информацию, характеризующую выполняемый код. При объектно-ориентированном подходе добавить эту функциональность в существующий код не так-то просто, поэтому операция хронометража часто добавляется в интересующие методы и затем, когда становится не нужна, комментируется (мы надеемся!):

```
public class Foo
     public void expensiveComputation () {
          long start = System.currentTimeMillis();
          try {
               // выполнить вычисления
          } catch (Exception e) {
               // вывести информацию об ошибке
          } finally {
               long stop = System.currentTimeMillis();
               System.out.println("Run time: " + (stop - start) + "ms");
          }
     }
}
```

Для регистрации производительности метода требуется изменить его, и такие изменения придется внести много раз, во все интересующие нас методы. Но это слишком сложно, в общем случае хотелось

бы иметь возможность «обернуть» метод некоторой функциональностью. Такие функции или методы, образующие «обертки» иногда называют *советами* (advices).

С возможностью определять советы отдельно от методов, к которым они применяются, мы получаем весьма гибкий инструмент:

- советы можно выборочно применять[1] к отдельным методам, не изменяя их; советы, используемые для отладки и на этапе разработки (например, выполняющие хронометраж или трассировку) можно запретить полностью, изменив единственный конфигурационный параметр или просто отключив АОП-трансформацию;
- мы можем изменять поведение совета в одном месте, не рыская по всем методам, например, чтобы улучшить код, выполняющий хронометраж или определяющий другие характеристики, достаточно внести изменения в одном месте.

Одной из реализаций механизма АОП в Java является AspectJ (http://www.eclipse.org/aspectj/), расширение для Java, реализующее дополнительные Java-подобные конструкции поддержки АОП. В состав AspectJ входит собственный компилятор, компилирующий код AspectJ в Java-классы, которые затем могут вплетаться в классы приложения, исходя из определений (часто называются *срезами точек сопряжения* (pointcut)), указывающих какие методы и классы должны быть затронуты. Рассмотрим пример использования механизма AspectJ. Следующий класс содержит метод, производительность которого требуется измерить:

```
public class AspectJExample {
     public void longRunningMethod () {
          System.out.println("Starting long-running method");
          try {
               Thread.sleep((long)(1000 + Math.random() * 2000));
          } catch (InterruptedException e) {
          }
     }
}
```

Далее, определим аспект `Timing`, который будет применяться ко всем методам этого класса:

---

[1] Применение советов или других аспектных трансформаций часто называют *вплетением* (weaving).

```
public aspect Timing {
     pointcut profiledMethods(): call(* AspectJExample.* (..));                ❶

     long time;

     before(): profiledMethods() {                                             ❷
          time = System.currentTimeMillis();
     }

     after(): profiledMethods() {                                              ❸
          System.out.println("Call to " + thisJoinPoint.getSignature() +
               " took " + (System.currentTimeMillis() - time) + "ms");
     }
}
```

❶ AspectJ имеет свой синтаксис определения пакетов, классов и сигнатур методов; этот срез точек, которому соответствуют все методы в классе `AspectJExample`, будет использоваться для определения точек применения советов `before` и `after`.

❷ Код аспекта, выполняемый перед вызовом исследуемого метода. Получает текущее время, чтобы его можно был сопоставить с текущим временем после завершения метода.

❸ Код аспекта, выполняемый после вызова исследуемого метода. Он просто выводит продолжительность работы метода в `stdout`.

Если теперь запустить программу, вызывающую метод `AspectJExample.longRunningMethod`, на экране появятся следующие строки:

```
Starting long-running method
Call to void com.clojurebook.AspectJExample.longRunningMethod() took 1599ms
```

Вместо блуждания по всему приложению, чтобы изменить или убрать код, выполняющий «профилирование», достаточно изменить поведение совета AspectJ в одном месте или отключить его совсем, исправив срез точек сопряжения так, чтобы предотвратить вплетение аспектов в методы приложения.

Robert Hooke. Возможности аспектно-ориентированного программирования в Clojure легко можно реализовать с помощью переменных и функций первого порядка. Функции в Clojure легко могут передаваться в аргументах другим функциям, а переменные – переопределяться во время выполнения; комбинация этих двух особенностей позволяет «обертывать» функции другими функциями и изменять их поведение или результаты.

Библиотека Robert Hooke (https://github.com/technomancy/robert-hooke) поддерживает простой и мощный способ определения советов (называются ловушками (hooks)) для функций. Добавить в пример выше ловушку, определяющую время выполнения функции, совсем несложно:

```
(defn time-it [f & args]
  (let [start (System/currentTimeMillis)]
    (try
      (apply f args)
      (finally
        (println "Run time: " (- (System/currentTimeMillis) start) "ms")))))
```

`time-it` – это обычная функция на языке Clojure. Она принимает другую функцию `f` и некоторое множество аргументов `args`, которые будут переданы функции `f`. Наш совет может выполнять любые операции до и после вызова `f` (если он вообще решит вызвать `f`). В данном случае мы просто выводим время выполнения функции.

Для вызова `f` из `time-it` не нужно ничего изобретать – мы просто вызываем ее с помощью `apply` и передаем ей аргументы, полученные в `args`. Обратите внимание, что благодаря поддержке произвольного количества аргументов, `time-it` сможет работать с любыми функциями, независимо от количества их аргументов.

Ниже показано, как можно применить этот совет к функции:

```
(require 'robert.hooke)

(defn foo [x y]
  (Thread/sleep (rand-int 1000))
  (+ x y))

(robert.hooke/add-hook #'foo time-it)  ❶
```

❶ Функция `robert.hooke` добавит ловушку `time-it` для переменной `#'foo`, которая будет перехватывать все вызовы функции в этой переменной.

Если теперь вызвать `foo`, на экране появятся результаты выполнения `time-it`:

```
(foo 1 2)
; Run time: 772 ms
;= 3
```

Библиотека Robert Hooke также поддерживает возможность временно или постоянно отключать или удалять ловушки:

```
(robert.hooke/with-hooks-disabled foo (foo 1 2))      ❶
;= 3

(robert.hooke/remove-hook #'foo time-it)              ❷
;= #<user$foo user$foo@4f13f501>
(foo 1 2)
;= 3
```

❶ `with-hooks-disabled` позволяет временно отключить ловушки, привязанные к указанной переменной.
❷ `remove-hook` удалит указанную ловушку из заданной переменной.

Обратите внимание, что все эти манипуляции были выполнены в оболочке REPL. Добавление, удаление и временное отключение ловушек в переменных можно производить во время выполнения, используя критерии, наиболее подходящие для вашего приложения.

Наконец, так как ловушки добавляются и удаляются в обычных переменных и с помощью обычных функций (таких как `add-hook` и `remove-hook`), нет никакой необходимости использовать своеобразный синтаксис AspectJ определения срезов точек сопряжения, демонстрировавшийся выше. В Clojure имеются отличные инструменты интроспекции, с помощью которых можно выполнять такие операции, как добавление ловушки во все переменные, имеющиеся в пространстве имен:

```
(require 'clojure.set)
;= nil
(doseq [var (->> (ns-publics 'clojure.set)
                 (map val))]
  (robert.hooke/add-hook var time-it))
;= nil
(clojure.set/intersection (set (range 100000))
                          (set (range -100000 10)))
; Run time: 97 ms
;= #{0 1 2 3 4 5 6 7 8 9}
```

Ничего из этого не требует специального языка или поддержки компилятора, как при использовании AspectJ. Фактически библио-

тека Robert Hooke занимает около 100 строк простого кода на Clojure. Если бы ее не существовало, вы легко могли бы написать ее сами.

В заключение

Как было показано в этой главе, многие распространенные шаблоны проектирования оказываются тривиально простыми в реализации и в большинстве случаев просто скрываются в самом языке Clojure и его библиотеках. Благодаря отделению функций и данных, наряду с функциями первого порядка и инструментами, позволяющими объединять и комбинировать их, архитектуру приложения на языке Clojure обычно можно построить вообще без применения шаблонов проектирования, которые в большинстве своем предназначены, для борьбы с излишней сложностью объектно-ориентированных подходов, таких как воплощенные в Java и Ruby.

Глава 13. Тестирование

Концептуально тестирования в Clojure выполняется практически так же как в Java, Python или Ruby. Независимо от используемого языка, задачи всегда одни и те же:

1. Воссоздать соответствующее окружение.
2. Выполнить некоторый код.
3. Убедиться, что поведение кода и возвращаемые им значения соответствуют ожидаемым.

Конечно, каждая из этих задач может существенно отличаться, в зависимости от языка и используемого фреймворка тестирования. В этой главе мы исследуем приемы тестирования в Clojure, сосредоточившись на использовании фреймворка `clojure.test`, входящего в состав стандартной библиотеки языка.

Неизменяемые значения и чистые функции

В объектно-ориентированных языках, таких как Java, Python или Ruby, тестирование может оказаться сложной задачей. Объекты могут весьма тонко взаимодействовать друг с другом. Изменение одного объекта может приводить к изменению произвольного числа других объектов или поведение одного объекта может неявно зависеть от состояния других. Эти взаимодействия часто имеют комбинированную природу, что осложняет учет всех характеристик окружения, способных влиять на поведение программы, и, соответственно, их тестирование.

Как описывалось в главе 2, Clojure потворствует использованию неизменяемых значений и чистых функций. Код, написанный с использованием этих особенностей, получается особенно простым, с точки зрения тестирования: если результат функции определяется только ее аргументами, для надежного охвата тестированием

достаточно будет простых модульных тестов. Разумеется, интеграционные и функциональные тесты остаются важной частью приложения и необходимы для проверки других функций, не являющихся чистыми.

Создание фиктивных значений

В объектно-ориентированных языках главным инструментом тестов являются *фиктивные* объекты (mock objects), имитирующие поведение реальных объектов или служб, от которого зависит некоторый код. Благодаря применению фиктивных объектов, тестируемый код можно более надежно изолировать от влияния на результаты тестов возможных ошибок в зависимостях или их переменчивого поведения.

Создание фиктивных объектов может оказаться неожиданно сложным делом, особенно в языках со статической системой типов. Специально для создания фиктивных объектов может потребоваться определить интерфейсы, а также специальные типы данных, чтобы иметь возможность представления однородного API к данным, получаемым от действующей службы или от ее имитации.

При тестировании в Clojure фиктивные объекты редко бывают нужны. Благодаря неизменяемым коллекциям и записям надобность в фиктивных данных просто отпадает сама собой; вы можете извлечь данные из действующего окружения, сохранить их и использовать в тестах, без всяких оберток, без преобразований любого вида и прочих гимнастических упражнений[1]. «Фиктивные данные» в Clojure – это всего лишь... данные.

Одним из следствий применения фиктивных объектов в других языках является необходимость имитировать функции Clojure. Рассмотрим функцию, отыскивающую адрес пользователя по его имени:

---

[1] Иногда в процессе тестирования возникает потребность в искусственно *сгенерированных* тестовых данных – идея, получившая некоторое распространение, благодаря фреймворку QuickCheck для Haskell – но такие данные остаются неизменяемыми и обладают всеми общими признаками данных в Clojure. Существует множество библиотек на языке Clojure, поддерживающих возможность искусственного создания данных, включая `test.generative` (https://github.com/clojure/test.generative/), ClojureCheck (https://bitbucket.org/kotarak/clojurecheck) и `re-rand` (https://github.com/weavejester/re-rand).

```
(defn get-address
  [username]
  ;; обращение к базе данных
  )
```

Эта функция будет терпеть неудачу везде, где база данных не доступна или не была настроена должным образом. А тесты, вовлекающие в работу эту функцию, будут терпеть неудачу, если база данных *доступна*, но содержит данные, отличающиеся от тех, что ожидают тесты.

Одним из возможных решений этой проблемы является функция `with-redefs`. Она временно замещает корневые значения (root values) указанных переменных некоторыми другими значениями, выполняет их и восстанавливает оригинальные корневые значения. Фактически, она выполняет подделку переменных:

```
(with-redefs [address-lookup (constantly "123 Main St.")]
  (println (address-lookup)))
; 123 Main St.
```

Практически точно так же можно использовать форму `binding` (описывается в разделе «Динамическая область видимости» в главе 4), но во многих случаях функция `with-redefs` оказывается удобнее, особенно при тестировании:

- `with-redefs` не накладывает ограничений на переменные, которые могут подвергаться ее воздействию, в отличие от формы `binding`, которая работает только с переменными, помеченными как `^:dynamic`;
- изменения, которые выполняет `with-redefs`, не являются локальными для потока выполнения, то есть, все потоки, агенты, объекты `future` и прочее, будут наблюдать временное корневое значение переменной;
- динамические переменные, затронутые вызовом формы `binding`, могут изменяться с помощью `set!`; это не препятствует использованию `alter-var-root` с переменными, временно измененными с помощью `with-redefs`, но недоступность `set!`, когда она не нужна, может оказаться разумной мерой предосторожности.

Проще говоря, динамические переменные и `binding` часто оказываются слишком мощными инструментами, чем необходимо для те-

стирования, предоставляющими возможности, которые могут быть нежелательны в этом контексте.

В любом случае `with-redefs` и `binding` можно использовать совместно с механизмом креплений (fixture facility), поддерживаемым фреймворком `clojure.test` и гарантирующим подготовку переменных для тестирования, чтобы обеспечить возврат функциями известных постоянных значений или для перенастройки операций, например, на использование определенной базы данных с помощью локальных настроек тестов. Подробнее о механизме креплений будет рассказываться в разделе «Крепления (fixtures)» ниже.

clojure.test

`clojure.test` — «официальный» фреймворк тестирования для Clojure. Это простая библиотека, но ее вполне достаточно для решения многих задач.

> **Примечание.** Существуют и другие распространенные фреймворки тестирования для Clojure, поддерживающие более сложную семантику и обладающие более широкими возможностями. Наиболее популярным из них является Midje, доступный по адресу: https://github.com/marick/Midje.

Контрольные проверки (assertions) при использовании `clojure.test` реализуются с помощью макроса `is`. Он вычисляет единственное выражение, проверяет логическую истинность результата и возвращает значение выражения. Макрос `is` сообщает о всех неудачах, возвращая сообщение (если было указано) и фактически полученные значения:

```
(use 'clojure.test)

(is (= 5 (+ 4 2)) "I never was very good at math...")
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)
; I never was very good at math...
; expected: (= 5 (+ 4 2))
;    actual: (not (= 5 6))
;= false

(is (re-find #"foo" "foobar"))
;= "foo"
```

Макрос `is` определяет множество особых проверок для использования в выражениях[1]. Например, `thrown?` проверит, было ли возбуждено исключение определенного типа в ходе вычисления выражения:

```
(is (thrown? ArithmeticException (/ 1 0)))                          ❶
;= #<ArithmeticException java.lang.ArithmeticException: Divide by zero>
(is (thrown? ArithmeticException (/ 1 1)))
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)
; expected: (thrown? ArithmeticException (/ 1 1))
;   actual: nil
;= nil
```

❶ Когда условия проверки `thrown?` выполняются, `is` возвращает исключение.

`thrown-with-msg?` — похожая проверка, но она дополнительно проверяет соответствие текста сообщения об ошибке указанному регулярному выражению:

```
(is (thrown-with-msg? ArithmeticException #"zero" (/ 1 0)))
;= #<ArithmeticException java.lang.ArithmeticException: Divide by zero>
(is (thrown-with-msg? ArithmeticException #"zero" (inc Long/MAX_VALUE)))
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)
; expected: (thrown-with-msg? ArithmeticException
; #"zero" (inc Long/MAX_VALUE))                                     ❶
;   actual: #<ArithmeticException java.lang.ArithmeticException:
;   integer overflow>
;= #<ArithmeticException java.lang.ArithmeticException: integer overflow>
```

❶ Когда оператор, не выполняющий автоматическое расширение типа, вызывает переполнение, Clojure возбуждает исключение `ArithmeticException`; текст сообщения для этого исключения не содержит `#"zero"`, поэтому проверка терпит неудачу.

Тесты можно документировать и дополнять отчеты об ошибках с помощью макроса `testing`, который включает описание теста в текст сообщения:

---

[1] Также поддерживается возможность добавления своих проверок, посредством определения новых методов для мультиметода `clojure.test/assertexpr`. Подробности смотрите в документации для пространства имен `clojure.test`: http://clojure.github.com/clojure/clojure.test-api.html.

```
(testing "Strings"
  (testing "regex"
    (is (re-find #"foo" "foobar"))
    (is (re-find #"foo" "bar")))
  (testing ".contains"
    (is (.contains "foobar" "foo"))))
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)
; Strings regex
; expected: (re-find #"foo" "bar")
;   actual: (not (re-find #"foo" "bar"))
```

Определение тестов

Существует два способа определения тестов. Первый и, пожалуй, самый удобный – в виде определения автономной функции с помощью макроса `deftest`. Этот макрос просто определяет функцию без аргументов (подобно форме `defn`) и присоединяет к ней некоторые метаданные, позволяющие определить, что это тест. Во всем остальном тесты, созданные с помощью `deftest`, идентичны обычным функциям и могут вызываться из оболочки REPL:

```
(deftest test-foo
  (is (= 1 1)))
;= #'user/test-foo
(test-foo)
;= nil
```

Общепринято определять тесты в разных пространствах имен, отдельно от тестируемой библиотеки или приложения, обычно в подкаталоге `test` в дереве каталогов проекта. Как было описано в главе 8, популярные инструменты сборки для Clojure могут заглядывать в этот подкаталог и автоматически запускать тесты.

Все тесты, основанные на `clojure.test`, фактически определяются как переменные, где функции хранятся в слоте `:test` метаданных. В этом можно убедиться, взглянув на содержимое теста, объявленного выше:

```
(:test (meta #'test-foo))
;= #<user$fn__366 user$fn__366@4e842e74>
```

Функция (`test-foo`) просто делегирует выполнение операций функции в слоте `:test`. Такая организация может показаться стран-

ной, но она позволяет, например, связывать тесты с тестируемыми функциями с помощью макроса with-test[1]. Этот макрос принимает в первом аргументе любую форму определения переменной и произвольное число других форм, которые будут составлять тело функции-теста в слоте :test:

```
(with-test
  (defn hello [name]
    (str "Hello, " name))
  (is (= (hello "Brian") "Hello, Brian"))
  (is (= (hello nil) "Hello, nil")))
;= #'user/hello
```

Определив функцию hello таким способом, мы можем использовать ее в нашем приложении, при этом никакие тесты выполняться не будут:

```
(hello "Judy")
;= "Hello, Judy"
```

Тело формы with-test, следующее за формой определения переменной, будет упаковано в виде функции в слоте :test в метаданных этой переменной – это и есть тестовая функция:

```
((:test (meta #'hello)))
; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:5)
; expected: (= (hello nil) "Hello, nil")
;   actual: (not (= "Hello, " "Hello, nil"))
;= false
```

К счастью функция clojure.test/run-tests автоматически использует эти метаданные для поиска тестов в одном или нескольких пространствах имен и выполняет их все. То есть, она может находить тесты как в загруженном исходном файле, так и тесты, объявленные в оболочке REPL:

```
(run-tests)     ❶
; Testing user
```

[1] Этот прием можно рассматривать, как более структурированную версию модуля doctest в Python или rubydoctest в Ruby, позволяющую хранить тесты рядом с определениями тестируемых функций.

```
;
; FAIL in (hello) (NO_SOURCE_FILE:5)
; expected: (= (hello nil) "Hello, nil")
;   actual: (not (= "Hello, " "Hello, nil"))
;
; Ran 2 tests containing 3 assertions.
; 1 failures, 0 errors.
;= {:type :summary, :pass 2, :test 2, :error 0, :fail 1}
```

❶ Если пространство имен не указано, `run-tests` выполняет поиск переменных с функциями `:test` в пространстве имен `*ns*`.

В отличие многих компилирующих языков, использующих цикл сборка-компиляция-тестирование, Clojure упрощает тестирование функций, написанных с использованием таких утилит, как `run-tests`.

Один из недостатков такого подхода состоит в том, что однажды объявленные тесты продолжают существовать на протяжении всего жизненного цикла JVM. Загрузка файла с диска или подключение пространства имен с тестами не приводит к уничтожению или переопределению тестов, то есть нежелательные тесты никуда не исчезнут и `run-tests` будет находить их и выполнять, пока вы не перезапустите JVM – чего мы стараемся избегать по мере возможности. К счастью с помощью `ns-unmap` можно удалить переменную (и ее тест)[1]:

```
(ns-unmap *ns* 'hello)
;= nil
(run-tests)
; Testing user
;
; Ran 1 tests containing 1 assertions.
; 0 failures, 0 errors.
;= {:type :summary, :pass 1, :test 1, :error 0, :fail 0}
```

Другое решение заключается в том, чтобы изменить метаданные функции `hello`, удалив из него функцию `:test`. В случае с переменными, объявленными с помощью `with-test`, этот прием позволит продолжать использовать основную функцию:

---

[1] Подробное описание `ns-unmap` и других функций интроспекции пространств имен можно найти в разделе «Определение и использование пространств имен», в главе 8, и в разделе «Интроспекция пространств имен», в главе 10.

```
(with-test
  (defn hello [name]
    (str "Hello, " name))
  (is (= (hello "Brian") "Hello, Brian"))
  (is (= (hello nil) "Hello, nil")))
;= #'user/hello
(alter-meta! #'hello dissoc :test)
;= {:ns #<Namespace user>, :name hello, :arglists ([name]),
;=  :line 2, :file "NO_SOURCE_PATH"}
(run-tests *ns*)
; Testing user
;
; Ran 1 tests containing 1 assertions.
; 0 failures, 0 errors.
;= {:type :summary, :pass 1, :test 1, :error 0, :fail 0}
(hello "Rebecca")
;= "Hello, Rebecca"
```

«Комплекты» тестов

Внутри тестов без всяких проблем можно вызывать другие тестовые функции:

```
(deftest a
  (is (== 0 (- 3 2))))
;= #'user/a
(deftest b (a))
;= #'user/b
(deftest c (b))
;= #'user/c
(c)
; FAIL in (c b a) (NO_SOURCE_FILE:2)    ❶
; expected: (== 0 (- 3 2))
;   actual: (not (== 0 1))
```

❶ Отчет об ошибке включает в «стек» тестовую функцию, вызвавшую ошибку; стеком здесь является список `(c b a)`[1].

[1] К сожалению, форма записи определяет цепочку вызовов – `c` вызвала `b`, которая вызвала `a` – а не единственный вызов, то есть, попытка вычислить `(c b a)` потерпит неудачу, потому что функции, объявленные через `deftest`, не принимают аргументов, в том числе и функция с именем `c`.

Определение «комплектов» тестов таким способом имеет определенные удобства, но идет вразрез с поведением по умолчанию функции `run-tests` (и, соответственно, вразрез с функциональностью тестирования в инструментах сборки Clojure): поскольку механизм тестирования вызывает функции `:test` в метаданных всех переменных, он выполнит все тестовые функции непосредственно, даже те, что являются частью «комплектов». В лучшем случае это просто увеличит время тестирования из-за избыточных вызовов тестовых функций; в худшем – вы получите множество повторяющихся сообщений об ошибках:

```
(run-tests)
; Testing user
;
; FAIL in (b a) (NO_SOURCE_FILE:2)
; expected: (== 0 (- 3 2))
;    actual: (not (== 0 1))
;
; FAIL in (c b a) (NO_SOURCE_FILE:2)
; expected: (== 0 (- 3 2))
;    actual: (not (== 0 1))
;
; FAIL in (a) (NO_SOURCE_FILE:2)
; expected: (== 0 (- 3 2))
;    actual: (not (== 0 1))
;
; Ran 6 tests containing 3 assertions.
; 3 failures, 0 errors.
;= {:type :summary, :pass 0, :test 6, :error 0, :fail 3}
```

Эта проблема имеет два решения. Первое: в каждом пространстве имен можно определить главную точку входа для `run-tests`; эта точка входа должна быть функцией без аргументов с именем `test-ns-hook`. При наличии `test-ns-hook`, в данном пространстве имен вызываться будет только эта функция. Этот подход позволяет определять, какие тесты будут выполняться и в каком порядке:

```
(defn test-ns-hook [] (c))
;= #'user/test-ns-hook
(run-tests)
; Testing user
;
```

```
; FAIL in (c b a) (NO_SOURCE_FILE:2)
; expected: (== 0 (- 3 2))
;    actual: (not (== 0 1))
;
; Ran 3 tests containing 1 assertions.
; 1 failures, 0 errors.
;= {:type :summary, :pass 0, :test 3, :error 0, :fail 1}
```

Второе решение: включить второстепенные проверки в обычные функции. У них не будет слота `:test` в метаданных и потому они не будут обнаруживаться и вызываться функцией `run-tests`:

```
(ns-unmap *ns* 'test-ns-hook)          ❶
;= nil
(defn a
  []
  (is (== 0 (- 3 2))))
;= #'user/a
(defn b [] (a))
;= #'user/b
(deftest c (b))
;= #'user/c
(run-tests)
; Testing user
;
; FAIL in (c) (NO_SOURCE_FILE:3)       ❷
; expected: (== 0 (- 3 2))
;    actual: (not (== 0 1))
;
; Ran 1 tests containing 1 assertions.
; 1 failures, 0 errors.
;= {:type :summary, :pass 0, :test 1, :error 0, :fail 1}
```

❶ Для начала сбросим функцию `test-ns-hook`, объявленную прежде.

❷ Обратите внимание, что теперь «стек» теста содержит только одну функцию (`c`), а не три (`c b a`), как прежде. Это объясняется тем, что `run-tests` обнаруживает только тестовые функции и не замечает обычные, такие как `a` и `b`.

Крепления (fixtures)

Крепления (fixtures) дают возможность определять и удалять настройки служб и баз данных, фиктивные функции и тестовые данные, гарантируя выполнение всех тестов в пространстве имен

в границах управляемого контекста. Они напоминают методы `setUp` и `tearDown` (или аннотации `@Before` и `@After`) в библиотеках модульного тестирования xUnit, но обеспечивают практически неограниченный контроль над тестовым окружением.

Крепление (fixture) – это обычная функция высшего порядка, принимающая единственный аргумент – тест или множество тестов, которые должны выполняться в контексте, настраиваемом и удаляемом креплением. Крепления могут определяться в любом месте, повторно использоваться из других тестовых пространств имен, и в каждом пространстве имен могут определяться в любом количестве.

Применять крепления к пространствам имен можно двумя способами.

1. Крепления могут вызываться для каждого теста, найденного в пространстве имен. В этом случае для пространства имен, содержащего *n* тестов, крепление будет вызвано *n* раз, и каждый раз с функцией, соответствующей единственному тесту.
2. Крепления могут вызываться всего один раз для каждого пространства имен. Тогда контекст, настраиваемый креплением, будет применяться ко всем тестовым функциям в этом пространстве имен. Для пространства имен, содержащего *n* тестов, крепление будет вызвано только один раз, с функцией, которая вызовет все тестовые функции в этом пространстве имен.

В любом случае, реализации креплений имеют следующую структуру:

```
(defn some-fixture
  [f]
  (try
    ;; настройка соединений с базами данных, загрузка тестовых данных,
    ;; определение фиктивных функций с помощью `with-redefs` или `binding`,
    ;; и так далее.
    (f)
    (finally
      ;; закрытие соединений с базами данных, файлов, и так далее.
      )))
```

Какой бы вариант запуска крепления вы ни выбрали, для *каждого* теста в отдельности или для *всех* тестов сразу, поведение крепления всегда тесно связано с объектами тестирования и во многом зависит от особенностей тестового окружения.

Варианты *для всех* и *для каждого* отвечают большинству требований, однако крепления в `clojure.test` предлагают лишь часть гибкости, доступной при определении собственных функций `test-ns-hook`. В отличие от креплений, определяющих два варианта использования, функция `test-ns-hook` дает *полный* контроль над тем, что выполнять, в каком порядке, и какие операции производить до и после каждого теста. Крепления часто оказываются более удобными. Например, при их использовании нет нужды явно управлять поиском и вызовом тестовых функций в пространстве имен, чем приходится неявно заниматься при использовании `test-ns-hook`.

> **Внимание.** Крепления и функции `test-ns-hook` являются взаимоисключающими. Если вы определите функцию `test-ns-hook`, тогда крепления не будут использоваться.

Чтобы познакомиться с креплениями поближе, вернемся к функции `configured-petstore` из примера 12.2, повторяющейся ниже с определениями дополнительных записей и протокола:

```
(defprotocol Bark
  (bark [this]))

(defrecord Chihuahua [weight price]
  Bark
  (bark [this] "Yip!"))

(defrecord PetStore [dog])
  (defn configured-petstore
  []
  (-> "petstore-config.clj"
    slurp
    read-string
    map->PetStore))
```

Тест для `configured-petstore` очень прост; нам нужно лишь сравнить возвращаемый ею экземпляр `PetStore` с известным значением:

```
(def ^:private dummy-petstore (PetStore. (Chihuahua. 12 "$84.50")))

(deftest test-configured-petstore
  (is (= (configured-petstore) dummy-petstore)))
```

Функция `configured-petstore` загружает запись из файла `"petstore-config.clj"` в текущем каталоге. То есть этот тест будет терпеть неудачу в отсутствие файла или если он будет содержать данные, не соответствующие ожидаемому результату:

```
(run-tests)
; Testing user
;
; ERROR in (test-configured-petstore) (FileInputStream.java:-2)
; expected: (= (configured-petstore) dummy-petstore)
;   actual: java.io.FileNotFoundException: petstore-config.clj
     (No such file or directory)
;  at java.io.FileInputStream.open (FileInputStream.java:-2)
;      ...
;
; Ran 1 tests containing 1 assertions.
; 0 failures, 1 errors.
;= {:type :summary, :pass 0, :test 1, :error 1, :fail 0}
```

Мы должны обеспечить наличие требуемого файла, чтобы можно было проверить результат вызова `configured-petstore`. Для этого нужно крепление. У нас уже есть ожидаемый экземпляр `PetStore` (хранящийся в `dummy-petstore`), поэтому написать функцию-крепление, сохраняющую в файл читаемое представление записи и ее полей для `configured-petstore` будет совсем несложно:

```
(defn petstore-config-fixture
  [f]
  (let [file (java.io.File. "petstore-config.clj")]
    (try
      (spit file (with-out-str (pr dummy-petstore)))      ❶
      (f)                                                 ❷
      (finally
        (.delete file)))))                                ❸
```

❶ Здесь выполняется запись читаемого представления `dummy-petstore` в файл `"petstore-config.clj"`. Обратите внимание, что здесь используется функция `pr` вместо `print` или `println`, которые выводят данные в формате, удобочитаемом для человека, тогда как функции `pr` и `prn` выводят данные в виде, читаемом для Clojure.

❷ `f` – это либо тестовая функция, либо функция, вызывающая тестовые функции в пространстве имен. Что это будет за функция, зависит от того, как зарегистрировано крепление – как выполняемое однократ-

но (:once) для пространства имен, или для каждой (:each) тестовой функции в пространстве имен.

❸ Хорошее правило – максимально устранять эффект применения крепления. Если этого не сделать, файл "petstore-config.clj" будет сохраняться между запусками тестов, возможно, позволяя другим тестам выполняться успешно по ошибке.

Определив крепление[1], его следует зарегистрировать. Для тестирования мы будем использовать жизненный цикл :once:

```
(use-fixtures :once petstore-config-fixture)
```

Зарегистрировав крепление, можно быть уверенными, что configured-petstore будет вызвана с известными данными и тест выполнится успешно:

```
(run-tests)
; Testing user
;
; Ran 1 tests containing 1 assertions.
; 0 failures, 0 errors.
;= {:type :summary, :pass 1, :test 1, :error 0, :fail 0}
```

Точно так же функцию petstore-config-fixture можно было бы зарегистрировать как крепление, вызываемое для каждого (:each) теста, чтобы обеспечить ее выполнение для каждой тестовой функции, имеющейся в пространстве имен.

Расширение HTML DSL

В качестве практического примера тестирования в процессе разработки рассмотрим тесты для библиотеки, генерирующей разметку HTML. В конечном итоге у нас получится бесхитростная версия библиотеки Hiccup (https://github .com/weavejester/hiccup), воспроизводящей разметку HTML, которая отражает структуру коллекций Clojure.

Наша цель – дать возможность писать такие фрагменты:

---

[1] Технически крепления не обязательно должны определяться как переменные верхнего уровня; функции use-fixtures можно передавать даже крепления, оформленные как функции-литералы.

```
[:html
 [:head [:title "Propaganda"]]
 [:body [:p "Visit us at "
          [:a {:href "http://clojureprogramming.com"}
           "our website"]
          "."]]]
```

И компилировать их в разметку HTML:

```
<html>
  <head><title>Propaganda</title></head>
    <body>
      <p>Visit us at <a href="http://clojurebook.com">our website</a>.</p>
    </body>
</html>
```

Здесь векторы (в действительности любые упорядоченные коллекции) представляют элементы HTML, где первое значение является именем элемента, второе (необязательное) – ассоциативным массивом с атрибутами, а все остальные определяют содержимое элемента, которое может быть строками или другими векторами, представляющими дочерние элементы.

Напишем для начала несколько тестов. Мы знаем, что наша функция, генерирующая разметку HTML, будет чистой функцией. Она будет принимать некоторые исходные данные и возвращать результат, независимо от состояния внешнего окружения и не оказывая влияния на него. Нам не хотелось бы писать `(is (= expected (f input)))` снова и снова, поэтому большинство наших тестов будут выглядеть так:

```
(deftest test-addition
  (are [x y z] (= x (+ y z))
    10 7 3
    20 10 10
    100 89 11))
```

`are` – это вспомогательный макрос в `clojure.test`, позволяющий определять шаблоны проверок. Например, форма выше будет развернута в следующие строки:

```
(do
  (clojure.test/is (= 10 (+ 7 3)))
  (clojure.test/is (= 20 (+ 10 10)))
  (clojure.test/is (= 100 (+ 89 11))))
```

Макрос `are` помогает уменьшить количество повторений каждой проверки, но нам все еще необходимо повторять преобразование, например `(= expected (f input))`. Избежать этого нам поможет коротенький макрос:

```
(defmacro are* [f & body]
  `(are [x# y#] (~'= (~f x#) y#)
     ~@body))
```

Теперь у нас есть возможность писать тесты, как показано ниже:

```
(deftest test-tostring
  (are* str
    10 "10"
    :foo ":foo"
    "identity" "identity"))
```

Допустим, что имеется функция `html`, принимающая упорядоченную коллекцию и возвращающая строку с разметкой HTML. Предположим также, что имеется вспомогательная функция `attrs`, возвращающая строки с атрибутами для включения в элементы HTML. Мы будем конструировать эти функции постепенно, на каждом шаге удовлетворяя ожидания, заложенные в тестах.

Для начала вполне достаточно будет следующих тестов:

```
(require 'clojure.string)

(declare html attrs)

(deftest test-html
  (are* html
    [:html]
    "<html></html>"

    [:a [:b]]
    "<a><b></b></a>"

    [:a {:href "/"} "Home"]
    "<a href=\"/\">Home</a>"

    [:div "foo" [:span "bar"] "baz"]
    "<div>foo<span>bar</span>baz</div>"))

(deftest test-attrs
```

```
  (are* (comp clojure.string/trim attrs)    ❶
    nil ""

    {:foo "bar"}
    "foo=\"bar\""

    (sorted-map :a "b" :c "d")
    "a=\"b\" c=\"d\""))
```

❶ Здесь мы немного схитрили; функция `trim` позволит игнорировать начальные и конечные пробельные символы, добавленные функцией `attrs`, которые, как мы знаем, могут оказаться удобными в некоторых случаях.

Теперь перейдем к коду. Первая попытка:

```
(defn attrs
  [attr-map]
  (->> attr-map
    (mapcat (fn [[k v]] [k " =\"" v "\""]))
    (apply str)))

(defn html
  [x]
  (if-not (sequential? x)
    (str x)
    (let [[tag & body] x
          [attr-map body] (if (map? (first body))
                            [(first body) (rest body)]
                            [nil body])]
      (str "<" (name tag) (attrs attr-map) ">"
           (apply str (map html body))
           "</" (name tag) ">"))))
```

Посмотрим, что выявит тестирование.

```
(run-tests)
; Testing user
;
; FAIL in (test-html) (NO_SOURCE_FILE:6)
; expected: (= (html [:a {:href "/"} "Home"]) "<a href=\"/\">Home</a>")
;   actual: (not (= "<a:href =\"/\">Home</a>" "<a href=\"/\">Home</a>"))
;
; FAIL in (test-attrs) (NO_SOURCE_FILE:20)
```

```
; expected: (= ((comp clojure.string/trim attrs) {:foo "bar"}) "foo=\"bar\"")
;   actual: (not (= ":foo =\"bar\"" "foo=\"bar\""))
;
; FAIL in (test-attrs) (NO_SOURCE_FILE:20)
; expected: (= ((comp clojure.string/trim attrs)
;               (sorted-map :a "b" :c "d"))
;             "a=\"b\" c=\"d\"")
;   actual: (not (= ":a =\"b\":c =\"d\"" "a=\"b\" c=\"d\""))
;
; Ran 2 tests containing 7 assertions.
; 3 failures, 0 errors.
;= {:type :summary, :pass 4, :test 2, :error 0, :fail 3}
```

Ой... что не так? Похоже, что источником ошибок является функция `attrs`. В результатах можно видеть присутствие ключей Clojure, например, вызов `(attrs {:foo "bar"})` возвращает `":foo =\":bar\""`. Нам нужно вызвать `name` для этих ключевых слов, чтобы получить простые строки с именами атрибутов. Исправим проблему, а заодно уберем ненужный пробел между именем атрибута и знаком `=`:

```
(defn attrs
  [attrs]
  (->> attrs
    (mapcat (fn [[k v]] [(name k) "=\"" v "\""]))
    (apply str)))
```

Теперь перезапустим `test-attrs` и посмотрим, помогло ли исправление функции `attrs` избавиться от ошибок:

```
(test-attrs)
; FAIL in (test-attrs) (NO_SOURCE_FILE:20)
; expected: (= ((comp clojure.string/trim attrs)
;               (sorted-map :a "b" :c "d"))
;             "a=\"b\" c=\"d\"")
;   actual: (not (= ":a =\"b\":c =\"d\"" "a=\"b\" c=\"d\""))
```

Уже лучше, но, похоже, что мы забыли добавить пробел перед именем каждого атрибута. Исправим:

```
(defn attrs
  [attrs]
  (->> attrs
    (mapcat (fn [[k v]] [\space (name k) "=\"" v "\""]))
    (apply str)))
```

Что теперь говорят тесты?

```
(test-attrs)
;= nil
(run-tests)
; Testing user
;
; Ran 2 tests containing 7 assertions.
; 0 failures, 0 errors.
;= {:type :summary, :pass 7, :test 2, :error 0, :fail 0}
```

Отлично! Похоже, что теперь у нас есть простой и понятный код. Посмотрим, как он справится с нашим первоначальным примером[1]:

```
(html [:html
          [:head [:title "Propaganda"]]
          [:body [:p "Visit us at "
                   [:a {:href "http://clojurebook.com"}
                     "our website"]
                   "."]]])
;= "<html>
;=   <head><title>Propaganda</title></head>
;=   <body>
;=     <p>Visit us at <a href=\"http://clojurebook.com\">our website</a>.</p>
;=   </body>
;= </html>"
```

Самое интересное в этом предметно-ориентированном языке (DSL) заключается в том, что разметка HTML представлена обычными структурами данных Clojure, от начала и до конца. У нас есть отличные средства для работы с такими структурами данных, поэтому возможность определять разметку HTML в виде:

```
(html (list* :ul (for [author ["Chas Emerick" "Christophe Grand" "Brian Carper"]]
                   [:li author])))
;= "<ul>
;=     <li>Chas Emerick</li>
;=     <li>Christophe Grand</li>
;=     <li>Brian Carper</li>
;=   </ul>"
```

[1] Нет, функция `html` не поддерживает форматирование результата; мы отформатировали его сами, чтобы, чтобы аккуратно разместить его на книжной странице.

выглядит намного привлекательнее, чем использование API к HTML DOM или такое безобразие, как внедрение строк с разметкой HTML в код и интерполяция данных в них. Среди возможностей генерирования разметки HTML в Clojure на одном краю находятся решения, подобные этому (где библиотека Hiccup является канонической реализацией), а на другом – такие решения как библиотека Enlive (будет использоваться в примерах главы 16), представляющие более традиционную модель, где содержимое в формате HTML обычно помещается в файлы шаблонов, которые затем заполняются программой.

Использование контрольных проверок

Наши функции, генерирующие разметку HTML, прекрасно справляются со своей задачей, но кто-нибудь где-нибудь обязательно попробует нарушить условия API. Например, стандартное для Clojure представление разметки XML (производимой функциями парсером *clojure.xml*) выглядит так:

```
{:tag :a, :attrs {:href "http://clojure.org"}, :content ["Clojure"]}
```

Оно совершенно отличается от представления HTML на основе векторов. Ничего удивительного, если функция `html` не сможет справиться с этим ассоциативным массивом, поскольку совершенно не ожидает получить его:

```
(html {:tag :a, :attrs {:href "http://clojure.org"}, :content ["Clojure"]})
;= "{:content [\"Clojure\"], :attrs {:href \"http://clojure.org\"},
;=    :tag :a}"
```

Вот так штука! Она вернула строку! Впрочем, это неудивительно, потому что мы по своей лени рекурсивно вызываем `html` для обработки содержимого элементов векторов. Но данный результат явно не имеет практической ценности для пользователя, который (возможно, вполне оправданно) полагает, что с помощью функции `html` можно преобразовать структуру данных, полученную от парсера *clojure.xml*, в строку HTML. Это одна из ситуаций, когда следует сгенерировать ошибку как можно скорее.

Введем контрольную проверку (assertion). Она будет проверять условие и генерировать ошибку при его несоблюдении:

```
(defn attrs
  [attrs]
  (assert (or (map? attr-map)                                    ❶
              (nil? attr-map)) "attr-map must be nil, or a map")
  (->> attrs
    (mapcat (fn [[k v]] [\space (name k) "=\"" v "\""]))
    (apply str)))

(attrs "hi")
;= #<AssertionError java.lang.AssertionError:
;= Assert failed: attr-map must be nil, or a map
;= (or (map? attr-map) (nil? attr-map))>
```

❶ Для проверки типов аргументов мы использовали макрос `assert`, но ее можно использовать везде, где можно выразить некоторое постоянное условие.

Поскольку контрольные проверки возбуждают исключение, они могут оказаться не самым лучшим инструментом для использования в окончательной версии приложения, где может оказаться нежелательным платить снижением производительности за выполнение проверок. Clojure позволяет включать и выключать контрольные проверки, устанавливая переменную `*assert*` в значение `true` или `false`, соответственно.

Так как `assert` – это макрос, переменная `*assert*` должна устанавливаться перед компиляцией кода, где этот макрос используется. Когда переменная `*assert*` имеет значение `false`, вызовы `assert` удаляются из скомпилированного кода целиком и не оказывают влияния на производительность.

```
(set! *assert* false)   ❶
;= false
(defn attrs
  [attr-map]
  (assert (or (map? attr-map)
              (nil? attr-map)) "attr-map must be nil, or a map")
  (->> attr-map
    (mapcat (fn [[k v]] [\space (name k) "=\"" v "\""]))
    (apply str)))
;= #'user/attrs
(attrs "hi")
;= #<UnsupportedOperationException java.lang.UnsupportedOperationException:
```

```
;=   nth not supported on this type: Character>
(set! *assert* true)    ❷
;= true
```

❶ Изменить состояние переменой `*assert*` можно в любой момент: для отдельного пространства имен, для всего приложения в целом (через системное свойство или переменную окружения) или, как в данном примере, в оболочке REPL.

❷ Мы вернули `*assert*` в прежнее состояние `true` для поддержки примеров, следующих ниже.

Функция `attrs` выше скомпилирована с отключенными контрольными проверками, поэтому теперь мы получили более расплывчатое сообщение об ошибке «nth not supported on this type» («nth не поддерживается для этого типа»), из-за того что функция, переданная в вызов `mapcat` попыталась деструктурировать символы строки. С другой стороны, в надежно протестированном приложении (возможно с применением множества контрольных проверок, модульных и функциональных тестов) отключение контрольных проверок позволит избавиться от накладных расходов, связанных с ними.

Предусловия и постусловия

Контрольные проверки очень часто используются для проверки входных аргументов и результатов функций. Так как Clojure особый упор делает на использование чистых функций, входные и выходные данные многих функций действительно стоит проверять.

Форма `fn` (и ее производные, такие как `defn`) имеет непосредственную поддержку контрольных проверок входных данных в форме *предусловий* (preconditions) и выходных – в форме *постусловий* (postconditions). Предусловия выполняются перед телом функции, а постусловия – после, но перед тем, как возвращаемое значение будет передано вызывающей программе. Если проверка какого-либо условия дает ложное значение, генерируется ошибка. Этот механизм можно использовать, чтобы гарантировать соответствие аргументов и возвращаемых значений определенным критериям, не загромождая тело функции проверками и возбуждением исключений и повторно используя эти проверки во всем приложении.

Если первым значением в теле функции является ассоциативный массив с ключом `:pre` или `:post`, этот ассоциативный массив будет

интерпретироваться как выражение пред- или постусловия и будет развернут в вызовы `assert` во время компиляции функции[1].

Значениями ключей `:pre` и `:post` должны быть векторы, каждый элемент которых должен представлять отдельную контрольную проверку. В предусловиях допускается ссылаться на параметры функции. Возвращаемое значение функции доступно в постусловиях под именем `%`, подобно тому, как обозначается первый аргумент в функциях-литералах. Перепишем наши функции `attrs` и `html` с использованием пред- и постусловий:

```
(defn attrs
  [attr-map]
  {:pre [(or (map? attr-map)                                          ❶
             (nil? attr-map))]}
  (->> attr-map
    (mapcat (fn [[k v]] [\space (name k) "=\"" v "\""]))
    (apply str)))

(defn html
  [x]
  {:pre [(if (sequential? x)                                          ❷
             (some #(-> x first %) [keyword? symbol? string?])
             (not (map? x)))]                                         ❸
   :post [(string? %)]}                                               ❹
  (if-not (sequential? x)
    (str x)
    (let [[tag & body] x
          [attr-map body] (if (map? (first body))
                            [(first body) (rest body)]
                            [nil body])]
      (str "<" (name tag) (attrs attr-map) ">"
           (apply str (map html body))
           "</" (name tag) ">"))))
```

❶ Как и прежде мы требуем, чтобы функции `attrs` в аргументе передавалось либо значение `nil`, либо ассоциативный массив.

❷ Предусловие для `html` немного сложнее. Если аргумент является упорядоченной коллекцией, то первый аргумент должен быть строкой, ключевым словом или символом. Или...

[1] Если ассоциативный массив является единственным выражением в теле функции, он будет интерпретироваться как возвращаемое значение функции, а не как ассоциативный массив с пред- и постусловиями.

❸ ...чем угодно, но не ассоциативным массивом.
❹ Постусловие в функции `html` требует, только чтобы возвращаемое значение было строкой. Здесь можно было бы добавить дополнительные условия, например, проверять соответствие произведенной разметки HTML некоторому объявлению DTD, и так далее.

Эти условия помогут обнаруживать некоторые наиболее распространенные ошибки. Вернемся к случаю, когда кто-нибудь попробует сгенерировать разметку HTML на основе ассоциативного массива, созданного парсером *clojure.xml*:

```
(html {:tag :a, :attrs {:href "http://clojure.org"}, :content ["Clojure"]})
;= #<AssertionError java.lang.AssertionError:
;= Assert failed: (if (sequential? x)
;=                   (some (fn* [p1__843#] (-> x first p1__843#))
;=                         [keyword? symbol? string?])
;=                   (not (map? x)))>
```

Обратите внимание, что пред- и постусловия компилируются в вызовы `assert` в теле функции. Это означает, что проверку этих условий можно включать и выключать с помощью переменной `*assert*`, и что условия нельзя добавить, удалить или изменить после того, как функция будет скомпилирована[1].

[1] Обойти эту проблему можно с помощью библиотеки Trammel, обеспечивающей поддержку *контрактного программирования* на языке Clojure, позволяющего с использованием пред- и постусловий, наряду с записями и инвариантными типами, убедиться в корректном поведении программы: https://github.com/fogus/trammel.

Глава 14. Реляционные базы данных

Реляционные базы данных, составляющие основу разработки программного обеспечения, существуют уже несколько десятков лет. Редкая организация не использует реляционные базы данных для своих нужд и редко, какому программисту не приходится извлекать и сохранять данные в базе данных, хотя бы изредка. Язык Java обладает богатейшей и качественной поддержкой реляционных баз данных посредством JDBC. Благодаря своей близости к JVM, Clojure позволяет с легкостью использовать всю эту богатейшую поддержку.

В нашем распоряжении имеется множество вариантов организации взаимодействий с реляционными базами данных из Clojure. `clojure.java.jdbc` – простая, но достаточно мощная библиотека, действующая как тонкая прослойка между Clojure и JDBC. Korma – еще одна библиотека для Clojure, поддерживающая более характерный для Clojure интерфейс. И, наконец, если библиотеки для Clojure не соответствуют вашему стилю или если вы ищете возможность подмешивать код на Clojure в существующие приложения на Java, вы всегда можете задействовать одну из зрелых и надежных библиотек или фреймворков для Java. В этой главе, среди прочего, мы исследуем настройку и использование фреймворка Hibernate.

clojure.java.jdbc

Независимо от того, что вы будете использовать в своих программах, библиотеки Java или библиотеки для Clojure, операции с базами данных всегда будут выполняться посредством JDBC, низкоуровневого API доступа к реляционным базам данных для Java. Библиотека `clojure.java.jdbc` (https://github.com/clojure/java.jdbc) является оберткой вокруг JDBC, поэтому она считается самой простой в использовании из Clojure:

```
[org.clojure/java.jdbc "0.1.1"]
```

Если говорить в терминах зависимостей, вам необходимо связать JDBC и `clojure.java.jdbc` с драйвером JDBC, соответствующим используемой базе данных. Драйверы JDBC существуют практически для всех существующих баз данных, поэтому поиск драйвера, соответствующего вашим потребностям, не должен занять много времени. Ниже перечислены некоторые координаты Leiningen драйверов JDBC для наиболее популярных баз данных:

```
[org.xerial/sqlite-jdbc "3.7.2"]            ; SQLite
[mysql/mysql-connector-java "2.0.14"]       ; MySQL
[postgresql "9.0-801.jdbc4"]                ; PostgreSQL
```

Во всех наших примерах мы будем использовать драйвер для SQLite (http://sqlite.org). Поскольку SQLite является «встраиваемой» базой данных – то есть, механизм управления этой базой данных встраивается в приложение, файлы базы данных хранятся там, где вы укажете, и для ее использования не требуется создавать соединение с удаленным сервером баз данных – весь код, который будет показан ниже, будет работать автономно, не требуя настройки и запуска отдельной базы данных. Благодаря абстракции, поддерживаемой механизмом JDBC, программы смогут работать с такими базами данных, как MySQL, PostgreSQL, Oracle и другими, практически без изменений в исходном коде.

Добавив в проект `clojure.java.jdbc` и библиотеку JDBC-драйвера `org.xerial/sqlite-jdbc` (http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC), мы сможем заняться исследованием особенностей взаимоействий с базами данных из оболочки Clojure REPL.

Все операции, поддерживаемые `clojure.java.jdbc`, требуют наличия «настроек»:

```
(require '[clojure.java.jdbc :as jdbc])
;= nil
(def db-spec {:classname "org.sqlite.JDBC"
              :subprotocol "sqlite"
              :subname "test.db"})
;= #'user/db
```

Здесь `db-spec` – это ассоциативный массив с настройками для `clojure.java.jdbc`:

1. Местоположение драйвера JDBC (значение ключа `:classname`).
2. Настройки драйвера и соединения.

Каждый драйвер JDBC требует немного отличающиеся настройки. Например, настройки сеанса связи с MySQL выглядят примерно так:

```
{:classname "com.mysql.jdbc.Driver"
 :subprotocol "mysql"
 :subname "//localhost:3306/databasename"
 :username "login"
 :password "password"}
```

Настройки для `clojure.java.jdbc` можно указать непосредственно через интерфейс `javax.sql.DataSource`:

```
{:datasource datasource-instance
 :username "login"
 :password "password"}
```

...извлечь из JNDI:

```
{:name "java:/comp/env/jdbc/postgres"
 :environment {}} ; необязательные параметры инициализации JNDI
    javax.naming.InitialContext
```

...или, при использовании одной из многих популярных баз данных[1], использовать соглашение по определению параметров соединения в виде строки в стиле URI, как показано ниже:

```
"mysql://login:password@localhost:3306/databasename"
```

Если вы уже знакомы с особенностями использования JDBC, все эти способы должны быть вам знакомы; определение соответствующей строки соединения или настроек базы данных из строки соединения в стиле JDBC, не должно вызывать сложностей.

После определения настроек базы данных можно приступать к использованию `clojure.java.jdbc` API:

```
(jdbc/with-connection db-spec)
;= nil
```

[1] На момент написания этих строк, такой способ настройки соединения поддерживался для баз данных: PostgreSQL, MySQL, SQLite, HSQLDB и Derby.

Функция `with-connection` открывает соединение с базой данных. В отсутствие дополнительных операций в теле формы `with-connection`, соединение будет открыто и тут же закрыто – это дает удобную возможность проверки параметров соединения из REPL, потому что неверное имя пользователя, пароль или URL базы данных приведут к возбуждению исключения.

Все выражения внутри формы `with-connection` выполняются в контексте открытого соединения с базой данных. Когда поток управления выходит за пределы формы, соединение автоматически закрывается и все ресурсы, связанные с ним, освобождаются[1].

С помощью функции `create-table` можно создать таблицу с именем «authors» и определить в ней несколько столбцов. В качестве имен столбцов можно использовать ключевые слова или строки; строки более подходят для имен таблиц/столбцов, содержащих символы, которые нельзя включать в литералы ключевых слов:

```
(jdbc/with-connection db-spec
  (jdbc/create-table :authors
    [:id "integer primary key"]
    [:first_name "varchar"]
    [:last_name "varchar"]))
;= (0)
```

`insert-records` – это простая функция, вставляющая данные в базу и возвращающая последовательность ассоциативных массивов с ключами, сгенерированными для каждой добавленной записи. Ключи в ассоциативных массивах должны соответствовать именам столбцов в таблице:

```
(jdbc/with-connection db-spec
  (jdbc/insert-records :authors
    {:first_name "Chas" :last_name "Emerick"}
    {:first_name "Christophe" :last_name "Grand"}
    {:first_name "Brian" :last_name "Carper"}))
```

[1] Типичная идиома языка Clojure, это лишь один из примеров автоматического управления ресурсами. В Clojure можно найти множество подобных функций `with-*`, открывающих и закрывающих дескрипторы файлов, сетевые соединения и так далее. В разделе «with-open, прощай finally», в главе 9, вы найдете обсуждение `with-open`, наиболее часто используемой функции из семейства `with-*` в Clojure.

```
;= ({:last_insert_rowid() 1}
;= {:last_insert_rowid() 2}
;= {:last_insert_rowid() 3})
```

Для извлечения информации из базы данных используется функция `with-query-results`. Форма `doall` в примере ниже играет важную роль, но подробнее об этом будет рассказываться ниже, в разделе «Отложенные вычисления»:

```
(jdbc/with-connection db-spec
  (jdbc/with-query-results res ["SELECT * FROM authors"]
    (doall res)))
;= ({:id 1, :first_name "Chas", :last_name "Emerick"}
;= {:id 2, :first_name "Christophe", :last_name "Grand"}
;= {:id 3, :first_name "Brian", :last_name "Carper"})
```

Обратите внимание, что в этом примере мы имеем дело исключительно с родными для Clojure типами данных. Имена таблицы и столбцов – ключевые слова; описание настроек базы данных – ассоциативный массив; эти же ассоциативные массивы используются для добавления данных в базу, и при извлечении данных, они возвращаются в ассоциативном массиве. Использование нескольких универсальных типов данных со множеством функций, оперирующих ими, полностью соответствует философии Clojure[1]. Обращение к полям в каждой строке набора данных, полученного из базы, – это обычная операция поиска в ассоциативном массиве. Например, мы можем воспроизвести полное имя каждого автора в таблице с помощью простой функции `map`.

Пример 14.1. Преобразование полученных данных с помощью функции map

```
(jdbc/with-connection db-spec
  (jdbc/with-query-results res ["SELECT * FROM authors"]
    (doall (map #(str (:first_name %) " " (:last_name %)) res))))
;= ("Chas Emerick" "Christophe Grand" "Brian Carper")
```

Обработку получаемых результатов можно выполнять с применением любых средств, имеющихся в Clojure.

---

[1] Знак вопроса (`?`) в запросе представляет параметр, а двойка (`2`) – значение, присваиваемое этому параметру.

Подробнее о with-query-results

Функция `with-query-results` – это основной инструмент извлечения информации из базы данных. Выражение `(with-query-results res query & body)` выполнит запрос к базе данных и затем выполнит `body`, в котором результаты запроса будут доступны в виде локальной привязки `res`. Сами результаты представляют собой «ленивую» последовательность ассоциативных массивов.

`with-query-results` поддерживает *параметризованные запросы*, распространенную особенность библиотек SQL, когда строка запроса является шаблоном, содержащим параметры запроса, значения которых передаются отдельно и интерполируются базой данных. Поддержка параметризованных запросов способствует многократному использованию запросов, что может увеличивать производительность запросов, выполняемых многократно, а также благотворно влияет на безопасность, в сравнении с более опасным приемом конструирования запросов путем конкатенации строк, открывающим возможность нападений вида «инъекция SQL» (SQL injection).

Запрос должен иметь вид вектора, первым элементом которого является строка SQL, а последующими элементами – значения параметров в строке запроса. Например:

```
(jdbc/with-connection db-spec
  (jdbc/with-query-results res ["SELECT * FROM authors WHERE id = ?" 2]  ❶
    (doall res)))
;= ({:id 2, :first_name "Christophe", :last_name "Grand"})
```

❶ Знак вопроса (?) в тексте запроса представляет параметр, а двойка (2) его значение.

Обратите внимание, что типы значений параметров запроса не имеют большого значения. Clojure, как динамический язык, обеспечит правильное внедрение указанных вами значений в SQL-запрос. Если передать значение недопустимого типа (которое не сможет быть преобразовано драйвером JDBC в значение нужного типа), механизм JDBC возбудит исключение.

Отложенные вычисления. Возможно вы заметили использование формы `doall` в сочетании с функцией `with-query-results` в примерах выше. Дело в том, что функция `with-query-results` возвращает результаты в виде «ленивой» последовательности[1], каждая запись реализу-

[1] См. раздел «Ленивые последовательности» в главе 3.

ется в ней только в случае явной необходимости. Это означает, что мы можем обрабатывать огромные массивы данных, возвращаемых запросами, не боясь исчерпать ресурсы, но мы не должны забывать, что источником этих данных является временная связь с базой данных. Взгляните на следующий, казалось бы простой, пример:

```
(jdbc/with-connection db-spec
  (jdbc/with-query-results res ["SELECT * FROM authors"]
    res))
;= ({:id 1, :first_name "Chas", :last_name "Emerick"})
```

Оппа! Запрос вернул всего одну строку, хотя должен был вернуть три. Проблема в том, что `with-query-results` возвращает результаты в виде «ленивой» последовательности. Оболочка REPL делает попытку вывести эту последовательность, которая в конечном итоге должна привести к реализации ее содержимого. Но эта попытка выполняется слишком поздно: соединение с базой данных к этому моменту уже будет закрыто (потому что поток управления выйдет из формы `with-connection`), и мы получим неполные результаты[1].

Решение состоит в том, чтобы извлечь все необходимые данные, пока соединение с базой данных остается открытым. Некоторые операции, такие как `reduce`, неявно вызывают реализацию последовательности; преобразование результатов с помощью других операций, также возвращающих «ленивые» последовательности, необходимо обертывать вызовом `doall`, как было показано в примере 14.1.

Если требуется всего лишь выполнить запрос и извлечь все полученные результаты, можно написать простую вспомогательную функцию:

```
(defn fetch-results [db-spec query]
  (jdbc/with-connection db-spec
    (jdbc/with-query-results res query
      (doall res))))
;= #'user/fetch-results
(fetch-results db-spec ["SELECT * FROM authors"])
;= ({:id 1, :first_name "Chas", :last_name "Emerick"}
;=  {:id 2, :first_name "Christophe", :last_name "Grand"}
;=  {:id 3, :first_name "Brian", :last_name "Carper"})
```

[1] Другие драйверы JDBC будут возбуждать исключение при попытке прочитать данные из закрытого соединения; драйвер SQLite автоматически извлекает первую запись, поэтому мы получили результат с одной строкой.

Транзакции

Чтобы выполнить последовательность операций с базой данных в рамках единой транзакции, достаточно просто завернуть код в форму `transaction`. Форма `transaction` принимает произвольное количество форм и выполняет их одну за другой, как части единой транзакции. Если в процессе выполнения будет возбуждено исключение или если какая-либо операция попытается нарушить ограничения целостности, устанавливаемые схемой базы данных, транзакция будет отменена. Если тело формы `transaction` выполнится без ошибок, транзакция будет подтверждена.

```
(jdbc/with-connection db-spec
  (jdbc/transaction
    (jdbc/delete-rows :authors ["id = ?" 1])
    (throw (Exception. "Abort transaction!"))))              ❶
;= ; Exception Abort transaction!
(fetch-results ["SELECT * FROM authors where id = ?" 1])
;= ({:id 1, :first_name "Chas", :last_name "Emerick"})       ❷
```

❶ Здесь мы возбуждаем исключение, чтобы принудительно прервать транзакцию.

❷ Данные в базе не изменились.

`transaction` – это макрос, реализующий все необходимые операции для запуска транзакции и гарантирующий откат транзакции в случае исключения. Он также отключает автоматическое подтверждение транзакций в объекте соединения JDBC и восстанавливает в первоначальное состояние по завершении выполнения тела формы `transaction`.

Допустим, что нам необходимо установить уровень изоляции транзакций для нашего соединения в значение `TRANSACTION_SERIALIZABLE`. Библиотека `clojure.java.jdbc` не поддерживает такую возможность непосредственно, но благодаря динамической природе Clojure, поддержке возможности взаимодействия с Java и следованию философии открытости данных, мы можем добиться желаемого результата.

`TRANSACTION_SERIALIZABLE` – это статический член класса `java.sql.Connection`, то есть из Clojure на него можно сослаться как `java.sql.Connection/TRANSACTION_SERIALIZABLE`. Доступ к динамической привязке объекта соединения внутри формы `with-connection` можно получить

с помощью функции `connection` из `clojure.java.jdbc`. Зная это, мы можем установить уровень изоляции транзакций, как показано ниже:

```
(jdbc/with-connection db-spec
  (.setTransactionIsolation (jdbc/connection)
    java.sql.Connection/TRANSACTION_SERIALIZABLE)
  (jdbc/transaction
    (jdbc/delete-rows :authors ["id = ?" 2])))
```

Пулы соединений

Функция `with-connection` проста в использовании, но по умолчанию она при каждом вызове открывает и закрывает новое соединение с базой данных. Это может существенно снижать производительность приложения.

Пул соединений – это кеш соединений с базой данных, которые могут повторно использоваться снова и снова. Многие серверы приложений поддерживают механизм на основе `DataSource` объединения соединений в пул, часто доступные через JNDI. В отсутствие сервера приложений можно использовать c3p0 (http://www.mchange.com/projects/c3p0) – популярную и легковесную библиотеку для создания пулов соединений в любых контекстах:

```
[c3p0/c3p0 "0.9.1.2"]
```

Добавив библиотеку c3p0 в зависимости проекта, вы сможете создать свою функцию, принимающую ассоциативный массив с настройками базы данных и возвращающую объект `DataSource`, представляющий пул соединений, созданный библиотекой c3p0:

```
(import 'com.mchange.v2.c3p0.ComboPooledDataSource)
; Feb 05, 2011 2:26:40 AM com.mchange.v2.log.MLog <clinit>
; INFO: MLog clients using java 1.4+ standard logging.
;= com.mchange.v2.c3p0.ComboPooledDataSource

(defn pooled-spec
  [{:keys [classname subprotocol subname username password] :as other-spec}]
  (let [cpds (doto (ComboPooledDataSource.)                                    ❶
               (.setDriverClass classname)
               (.setJdbcUrl (str "jdbc:" subprotocol ":" subname))
               (.setUser username)
               (.setPassword password))]
    {:datasource cpds}))
```

❶ Класс ComboPooledDataSource реализует DataSource (стандартный Java-интерфейс для соединений с любыми базами данных), поэтому он может использоваться с функцией with-connection.

Соединения в пуле инициализируются при первом обращении и сохраняются (с настройками, определенными для пула) для использования в следующих вызовах with-connection.

```
(def pooled-db (pooled-spec db-spec))
; Dec 27, 2011 8:49:28 AM com.mchange.v2.c3p0.C3P0Registry banner
; INFO: Initializing c3p0-0.9.1.2 [built 21-May-2007 15:04:56; debug? true;
; trace: 10]
;= #'user/pooled-db

(fetch-results pooled-db ["SELECT * FROM authors"])
; Dec 27, 2011 8:56:40 AM
; com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource
; getPoolManager
; INFO: Initializing c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource
; [ acquireIncrement -> 3, acquireRetryAttempts -> 30,
;   acquireRetryDelay -> 1000,...
;= ({:id 1, :first_name "Chas", :last_name "Emerick"}
;= {:id 2, :first_name "Christophe", :last_name "Grand"}
;= {:id 3, :first_name "Brian", :last_name "Carper"})

(fetch-results pooled-db ["SELECT * FROM authors"])
;= ({:id 1, :first_name "Chas", :last_name "Emerick"}
;= {:id 2, :first_name "Christophe", :last_name "Grand"}
;= {:id 3, :first_name "Brian", :last_name "Carper"})
```

Второй запрос в этом примере будет повторно использовать то же самое соединение, что и первый (о чем говорит отсутствие сообщения об инициализации в выводе второго вызова fetch-results). Настройки c3p0 по умолчанию подходят для большинства приложений, но не забывайте о богатстве параметров, поддерживаемых в локальных конфигурациях pooled-spec, позволяющих максимизировать пропускную способность.

Korma

Korma (http://sqlkorma.com) – это перспективный предметно-ориентированный язык для работы с реляционными базами данных в программах на Clojure. Его цель – обеспечить встроенную

поддержку взаимодействий с базами данных идиоматичным для Clojure способом; для этого он принимает на себя все хлопоты по автоматической генерации SQL для многих популярных баз данных и решению административных задач, таких как управление пулами соединений посредством c3p0. Знакомые с ActiveRecord в Ruby или аналогичными механизмами объектно-реляционного отображения (Object-Relational Mapping, ORM), найдут в Korma много знакомых черт, даже при том, что это – не фреймворк ORM.

Прежде чем использовать возможности Korma, необходимо добавить соответствующую зависимость в проект:

```
[korma "0.3.0"]
```

Вступление

Настроим несколько таблиц и добавим в них некоторые данные с помощью `clojure.java.jdbc` для дальнейшего использования.

```
(require '[clojure.java.jdbc :as jdbc])

(def db-spec {:classname "org.sqlite.JDBC"
              :subprotocol "sqlite"
              :subname "test.db"})

(defn setup
  []
  (jdbc/with-connection db-spec
    (jdbc/create-table :country
                       [:id "integer primary key"]
                       [:country "varchar"])
    (jdbc/create-table :author
                       [:id "integer primary key"]
                       [:country_id "integer constraint fk_country_id
                                     references country (id)"]
                       [:first_name "varchar"]
                       [:last_name "varchar"])
    (jdbc/insert-records :country
                         {:id 1 :country "USA"}
                         {:id 2 :country "Canada"}
                         {:id 3 :country "France"})
    (jdbc/insert-records :author
                         {:first_name "Chas"
                          :last_name "Emerick"
```

```
                        :country_id 1}
                       {:first_name "Christophe"
                        :last_name "Grand"
                        :country_id 3}
                       {:first_name "Brian"
                        :last_name "Carper"
                        :country_id 2}
                       {:first_name "Mark"
                        :last_name "Twain"
                        :country_id 1})))

(setup)
;= ({:id 1, :country_id 1, :first_name "Chas", :last_name "Emerick"}
;= {:id 2, :country_id 3, :first_name "Christophe", :last_name "Grand"}
;= {:id 3, :country_id 2, :first_name "Brian", :last_name "Carper"}
;= {:id 4, :country_id 1, :first_name "Mark", :last_name "Twain"})
```

Наши таблицы определены, как имеющие отношение «многие-к-одному» между авторами и странами. Настройка Korma на использование нашей базы данных выполняется просто:

```
(use '[korma db core])
(defdb korma-db db-spec)
```

`defdb` определяет соединение, которое будет использоваться механизмом Korma. Эта команда принимает те же аргументы, что содержатся в ассоциативном массиве с настройками для `clojure.java.jdbc`, поэтому здесь повторно используется ассоциативный массив `db-spec`.

Последнее соединение, настроенное с помощью формы `defdb`, играет роль соединения «по умолчанию», которое будет использоваться всеми запросами. Это удобно при работе с единственной базой данных, что на практике встречается чаще всего. Форма `defdb` также способна создать пул соединений с базой данных, что избавляет от необходимости выполнять эту работу вручную[1].

Следующий шаг в настройке Korma – определение *сущностей* (entities), хранящих характеристики таблиц базы данных. Сущности напоминают «модели» в Ruby ActiveRecord. В данном случае наши сущности могут выглядеть так:

---

[1] Получить соединение из такого пула можно с помощью функции `get-connection`, например, `(get-connection korma-db)`. Это дает возможность повторно использовать пул соединений Korma, если потребуется выполнить какие-то операции с помощью `clojure.java.jdbc`.

```
(declare author)

(defentity country
  (pk :id)
  (has-many author))

(defentity author
  (pk :id)
  (table :author)
  (belongs-to country))
```

Кроме всего прочего форма `defentity` определяет отношения между таблицами в базе данных.

Запросы

После определения характеристик таблиц, можно приступать к выполнению запросов:

```
(select author
  (with country)
  (where {:first_name "Chas"}))
;= [{:id 1, :country_id 1, :first_name "Chas",
;=   :last_name "Emerick", :id_2 1, :country "USA"}]
```

Макрос `select` – это предметно-ориентированная SQL-инструкция `SELECT`. `select` принимает различные функции, участвующие в конструировании запроса. Используемая здесь функция `with`, например, предписывает механизму Korma включить в запрос отношение, прежде описанное с помощью `defentity`. Обратите внимание, что в результаты включена пара ключ/значение `country`.

Более сложный запрос:

```
(select author
  (with country)
  (where (like :first_name "Ch%"))
  (order :last_name :asc)
  (limit 1)
  (offset 1))
;= [{:id 2, :country_id 3, :first_name "Christophe",
;=   :last_name "Grand", :id_2 3, :country "France"}]
```

`order`, `limit` и `offset` представляют соответствующие SQL-операторы. Гораздо больший интерес представляет функция `where`, которая сама

является реализацией миниатюрного предметно-ориентированного языка для конструирования SQL-предложений WHERE. Функция where способна обрабатывать весьма сложные определения условий:

```
(select author
      (fields :first_name :last_name)
      (where (or (like :last_name "C%")
                 (= :first_name "Mark"))))
;= [{:first_name "Brian", :last_name "Carper"}
;= {:first_name "Mark", :last_name "Twain"}]
```

Чтобы заглянуть за кулисы и увидеть SQL-код, генерируемый механизмом Korma, можно воспользоваться функцией sql-only:

```
(println (sql-only (select author
                     (with country)
                     (where (like :first_name "Ch%"))
                     (order :last_name :asc)
                     (limit 1)
                     (offset 1))))
;= ; SELECT "author".* FROM "author" LEFT JOIN "country"
;= ; ON "country"."id" = "author"."country_id"
;= ; WHERE "author"."first_name" LIKE ?
;= ; ORDER BY "author"."last_name" ASC LIMIT 1 OFFSET 1
```

Зачем использовать предметно-ориентированный язык?

Конечно, в Clojure можно использовать простые SQL-запросы, определяя их в виде строк с SQL-инструкциями. Фактически, именно такой подход поддерживает clojure.java.jdbc. Но подход на основе Korma имеет свои преимущества.

Строки с SQL-кодом не имеют четко выраженной структуры и управлять такими строками несколько сложнее. Например, как изменить строку запроса SELECT * FROM foo ORDER BY bar, чтобы отобрать конкретные поля, а не все (*)? Как добавить предложение WHERE? В конечном итоге можно дойти до того, что нам потребуется полноценный SQL-парсер.

Korma представляет запросы не как аморфные строки, а в виде ассоциативных массивов. Фактически, мы можем складывать эти запросы по кирпичикам, используя select* вместо select:

```
(def query (-> (select* author)
               (fields :last_name :first_name)
               (limit 5)))
;= #'user/query
```

Посмотрим, как выглядит этот ассоциативный массив `query`:

```
{:group [],
 :from
 [{:table "author",
   :name "author",
   :pk :id,
   :db nil,
   :transforms (),
   :prepares (),
   :fields [],
   :rel
   {"country"
    #<Delay@54f690e4:
      {:table "country",
       :alias nil,
       :rel-type :belongs-to,
       :pk {:korma.sql.utils/generated "\"country\".\"id\""},
       :fk
       {:korma.sql.utils/generated "\"author\".\"country_id\""}}>}}],
 :joins [],
 :where [],
 :ent
 {:table "author",
  :name "author",
  :pk :id,
  :db nil,
  :transforms (),
  :prepares (),
  :fields [],
  :rel
  {"country"
   #<Delay@54f690e4:
     {:table "country",
      :alias nil,
      :rel-type :belongs-to,
      :pk {:korma.sql.utils/generated "\"country\".\"id\""},
      :fk {:korma.sql.utils/generated "\"author\".\"country_id\""}}>},
      :limit 5,
```

```
       :type :select,
       :alias nil,
       :options nil,
       :fields (:last_name :first_name),
       :results :results,
       :table "author",
       :order [],
       :modifiers [],
       :db nil,
       :aliases #{}}
```

Чтобы изменить запрос, достаточно изменить лежащий в его основе ассоциативный массив, производимый формой `select*`, что и делают функции Korma, такие как `order`, `limit` и `offset`. Теперь, можно не определять фиксированные запросы целиком, а конструировать их последовательно и выполнять с помощью функции `exec`. Такой подход позволяет многократно использовать разные части запроса и инкапсулировать операции трансформации запросов в функции.

В последние версии ActiveRecord в Ruby on Rails (версия 3.0, к моменту написания этих строк) был добавлен очень похожий способ формирования SQL-запросов с использованием методов объектов запросов. В Ruby on Rails поддерживается возможность писать такой код:

```
employees = Person.where(:type => "employee")
# ... а потом ...
managers = employees.where(:role => "manager").order(:last_name)
managers.all.each do |e|
  ...
end
```

Реализация подобных операций с использованием Korma выглядит очень похожей:

```
(def employees (where (select* employees) {:type "employee"}))

;; ... а потом ...
(let [managers (-> employees
                   (where {:role "manager"})
                   (order :last_name))]
  (doseq [e (exec managers)]
    ; ... обработка результатов ...
    ))
```

Отложенные запросы. Запросы, сконструированные таким способом, являются «ленивыми», в том смысле, что данные из базы данных не будут извлекаться, пока запрос не будет выполнен явно вызовом функции `select`. Это несколько иной вид отложенных вычислений, чем в «ленивых» структурах данных языка Clojure. Возможно такие запросы правильнее было бы называть запросами, «выполняемыми по требованию».

Представьте, что имеется таблица с информацией о всех людях, когда-либо живших на земле. Мы могли бы указать, что данные из этой таблицы всегда должны извлекаться отсортированными по дате рождения. Затем, перед выборкой данных, мы можем сузить их диапазон с помощью предикатов и применения `LIMIT` и `OFFSET` для разбивки результатов на страницы.

```
(def humans (-> (select* humans)
                (order :date_of_birth)))                                    ❶

(let [kings-of-germany (-> humans
                           (where {:country "Germany" :profession "King"}))]  ❷
  (doseq [start (range 0 100 10)
          k (select kings-of-germany                                        ❸
              (offset start)
              (limit 10))]
    ...)
```

❶ Если бы запрос был выполнен в текущем его виде, мы получили бы в результате миллиарды записей. Однако `humans` определяет лишь порядок следования записей.

❷ В нужный момент времени мы можем определить дополнительные параметры запроса...

❸ ...и использовать такой запрос как основу для дальнейшего его усовершенствования, например, для добавления инструкций разбивки результатов на страницы, без необходимости вновь определять все критерии, накопленные в ассоциативном массиве `query`.

Hibernate

Если вам уже приходилось использовать Java или другой язык JVM для работы с реляционными базами данных, вы наверняка знакомы с Hibernate (http://www.hibernate.org), с одной из самых популярных Java-библиотек объектно-реляционного отображения. Одним и достоинств языка Clojure является простая возможность

использования библиотек и фреймворков для Java, и Hibernate в этом отношении не исключение.

Философия Hibernate совершенно отличается от философии языка Clojure: эта библиотека создает объекты, изменяет их и транслирует эти изменения в запросы к базе данных. Однако Clojure настолько гибкий язык, что позволяет использовать библиотеку Hibernate без лишних сложностей.

Настройка

Настроим библиотеку Hibernate, чтобы с ее помощью можно было создавать запрашивать и изменять таблицу `authors`, описанную выше в этой главе. Сначала необходимо добавить Hibernate в зависимости проекта:

```
[org.hibernate/hibernate-core "4.0.0.Final"]
```

В большинстве случаев работа с библиотекой Hibernate из Clojure связана с использованием объектов, уже сконструированных в Java[1]. Ниже демонстрируется Java-класс, представляющий имя автора с использованием аннотаций из Hibernate и JPA, указывающих, что класс представляет сущность, хранящуюся в базе данных, и определяющих типичное поведение автоинкрементного поля `id`:

```
package com.clojurebook.hibernate;

import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Entity;
import org.hibernate.annotations.GenericGenerator;

@Entity
public class Author {
     private Long id;
```

[1] Классы, наследующие класс `Entity` из Hibernate/JPA, с легкостью *можно* писать на языке Clojure. Другие примеры использования аннотаций Java-фреймворков в сочетании со средствами определения типов в Clojure можно найти в разделе «Аннотации», в главе 9. В случае применения Hibernate/JPA вам придется использовать форму `gen-class`, потому что для создания сущностей при выполнении запросов используется конструктор по умолчанию, не принимающий аргументов.

```
    private String firstName;
    private String lastName;

    public Author () {}

    public Author (String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    public Long getId () {
        return this.id;
    }
    public String getFirstName () {
        return this.firstName;
    }
    public String getLastName () {
        return this.lastName;
    }

    public void setId (Long id) {
        this.id = id;
    }
    public void setFirstName (String firstName) {
        this.firstName = firstName;
    }
    public void setLastName (String lastName) {
        this.lastName = lastName;
    }
}
```

Настройки для Hibernate обычно определяются не в ассоциативном массиве, как для `clojure.java.jdbc` или Korma, а в XML-файле *hibernate.cfg.xml*. В примере 14.2 приводится содержимое такого файла для настройки доступа к базе данных SQLite.

Пример 14.2. rsrc/hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration SYSTEM
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
```

```
  <session-factory>
    <property name="hibernate.connection.driver_class">
      org.sqlite.JDBC
    </property>
    <property name="hibernate.connection.url">
      jdbc:sqlite::memory:
    </property>
    <property name="hibernate.dialect">
      org.hibernate.dialect.HSQLDialect
    </property>
    <!-- Удалить и создать заново схему базы данных на запуске -->
    <property name="hbm2ddl.auto">create</property>
    <mapping class="com.clojurebook.hibernate.Author"/>
  </session-factory>
</hibernate-configuration>
```

Наконец, если для управления проектом используется инструмент Leiningen, нужно добавить лишь пару ключей в файл *project.clj*. Один – чтобы определить корневой каталог с исходным кодом на Java, который нужно компилировать (например, в *Author.java*), и другой – чтобы указать, где хранится файл *hibernate.cfg.xml file*:

```
:java-source-path "java"
:resources-path "rsrc"
```

В результате структура дерева каталогов проекта приобретает такой вид:

```
|-- project.clj
|-- rsrc
|    `-- hibernate.cfg.xml
|-- java
|    `-- com
|         `-- clojurebook
|              `-- hibernate
|                   |-- Author.java
```

Теперь можно скомпилировать наш Java-класс и запустить оболочку REPL, чтобы поэкспериментировать с Hibernate из Clojure:

```
% lein javac
...
% lein repl
```

Сначала необходимо импортировать требуемые Java-классы из Hibernate и наш новый класс `Author`:

```
(import '(org.hibernate.SessionFactory
          org.hibernate.cfg.Configuration
          com.clojurebook.hibernate.Author))
```

Библиотека Hibernate требует настроить фабричный объект сеанса, чтобы с его помощью открывать и закрывать сеансы работы с базой данных, и выполнять запросы. Очень важно, чтобы фабричный объект создавался в единственном экземпляре. В Java для этого можно было бы определить вспомогательный класс со статическим финальным полем, представляющим фабрику сеансов, и создавать экземпляр фабричного объекта в момент загрузки класса. Такой прием не считается в Java чем-то необычным и даже описывается в документации Hibernate (http://docs.jboss.org/hibernate/core/3.3/reference/en/html/tutorial.html):

```
public class HibernateUtil {
     private static final SessionFactory sessionFactory = buildSessionFactory();

     private static SessionFactory buildSessionFactory() {
          try {
               return new Configuration().configure().buildSessionFactory();
          }
          catch (Throwable ex) {
               System.err.println("Initial SessionFactory creation failed." + ex);
               throw new ExceptionInInitializerError(ex);
          }
     }

     public static SessionFactory getSessionFactory() {
          return sessionFactory;
     }
}
```

В Clojure можно использовать более простое решение – формы `defonce` и `delay`:

```
(defonce session-factory
  (delay (-> (Configuration.)
             .configure
             .buildSessionFactory)))
```

Как следует из имени, форма `defonce` действует подобно `def` и определяет переменную с некоторым значением, но оставляет прежде существовавшее определение нетронутым, даже при повторной загрузке пространства имен, содержащего определение фабрики сеансов. Форма `delay` гарантирует, что выражение `->`, которое создает и настраивает фабрику сеансов, не будет вычисляться до первого разыменования; это позволяет загружать или компилировать файлы с исходным кодом, не рискуя установить соединение с базой данных в этот момент.

Сохранение данных

Наш файл *hibernate.cfg.xml* содержит параметры настройки базы данных SQLite, хранящейся в памяти, поэтому в момент запуска оболочки REPL она будет пуста. Наполнить ее данными в Java можно так:

```
public static void saveAuthors (Author... authors) {
     Session session = sessionFactory.openSession();
     session.beginTransaction();
     for (Author author : authors) {
          session.save(author);
     }
     session.getTransaction().commit();
     session.close();
}

saveAuthors(new Author("Christophe", "Grand"),
               new Author("Brian", "Carper"), ...);
```

Ниже показана простая трансляция этого метода в функцию `add-authors` на Clojure:

Пример 14-3. add-authors

```
(defn add-authors
  [& authors]
  (with-open [session (.openSession @session-factory)]
    (let [tx (.beginTransaction session)]
      (doseq [author authors]
        (.save session author))
      (.commit tx))))

(add-authors (Author. "Christophe" "Grand") (Author. "Brian" "Carper")
  (Author. "Chas" "Emerick"))
```

Выполнение запросов

Теперь, когда мы обеспечили сохранность некоторых данных, попробуем извлечь список строк из таблицы `authors` и вывести имена авторов. В Java это можно сделать так:

```
Session session = HibernateUtil.getSessionFactory().openSession();
try {
     return (List<Author>)newSession.createQuery("from Author").list();
} finally {
     session.close();
}
```

Этот код очень просто переводится на язык Clojure:

Пример 14.4. get-authors

```
(defn get-authors
  []
  (with-open [session (.openSession @session-factory)]
    (-> session
      (.createQuery "from Author")
      .list)))
```

Конечно, в Clojure имеется больше гибких возможностей манипулирования данными, извлекаемыми из запроса Hibernate:

```
(for [{:keys [firstName lastName]} (map bean (get-authors))]    ❶
  (str lastName ", " firstName))
;= ("Carper, Brian" "Emerick, Chas" "Grand, Christophe")
```

❶ Функция `bean` преобразует Java-объект, являющийся компонентом JavaBean, в ассоциативный массив Clojure, ключи в котором совпадают с именами методов чтения, имеющихся в этом компоненте JavaBean.

Метод `.list` объекта запроса Hibernate возвращает экземпляр `java.util.ArrayList` с результатами. Выполнить обход элементов списка можно с помощью `doseq`, потому что Clojure гарантирует возможность преобразования любых экземпляров `java.util.List` в последовательности.

Избавление от шаблонного кода

Наш код на языке Clojure отлично работает, но его можно улучшить.

Обратите внимание на повторяющийся код в двух функциях. Открытие и закрытие сеанса, а также запуск и подтверждение транзакции – это операции, которые нам придется повторять снова и снова.

В Java, возможно, вы были бы вынуждены писать один и тот же код раз за разом, но в Clojure есть лучшее решение. Прежде всего, в Clojure можно использовать встроенный макрос `with-open`, позволяющий автоматически устанавливать соединение или открывать дескриптор, выполнять некоторый код и затем автоматически закрывать соединение или дескриптор. Поскольку сеансы в Hibernate закрываются вызовом стандартного метода `.close`, они автоматически поддерживаются макросом `with-open`.

Однако есть еще более удачный способ. Макрос `with-open` требует указывать имя для локального сеанса. Определив свой макрос поверх `with-open`, мы можем определить, что в теле этого макроса всегда будет использоваться имя по умолчанию `session`:

```
(defmacro with-session
  [session-factory & body]     ❶
  `(with-open [~'session (.openSession ~(vary-meta session-factory assoc
                                                   :tag 'SessionFactory))]          ❷
     ~@body))
```

❶ Первый аргумент, `session-factory`, – это форма для получения открытого сеанса. Все остальные аргументы – это формы, выполняемые в контексте открытого сеанса.

❷ Без указания типа `session-factory` компилятор не поймет, что `(.openSession factory#)` возвращает экземпляр Hibernate-класса `Session` и скомпилирует все последующие вызовы, использующие сеанс, так, что они будут использовать механизм рефлексии. Однако мы не можем указать тип для размаскированного символа `session-factory`, так как это указание будет относиться к форме `~session-factory`, а не к пользовательскому символу, связанному с `session-factory` внутри макроса. Чтобы исправить эту проблему, необходимо изменить метаданные символа, связанного с `session-factory`, чтобы тип значения определялся правильно. Обратите внимание, что значением `:tag` является *символ*, а не класс.

Вместо использования генератора символов для создания «безопасной» локальной привязки, которая не будет маскировать существующие привязки или «просачиваться» в пользовательский код внутри формы `with-session`, мы *хотим* связать экземпляр `Session` в данной области видимости с именем `session`. Для этого мы вы-

пускаем в обращение неквалифицированный (без указания пространства имен) символ, используя ~'session[1]. Голый символ session в этом месте в макросе вызвал бы ошибку компиляции, так как при развертывании макроса символ автоматически был бы квалифицирован текущим пространством имен, например, user/session, а такие имена, как мы знаем, нельзя использовать для локальных привязок.

С помощью этого макроса пример 14.4 можно записать более кратко:

```
(defn get-authors
  []
  (with-session @session-factory
    (-> session
      (.createQuery "from Author")
      .list)))
```

Сама по себе это *не особенно большая* экономия, но, умноженная на десятки, сотни или тысячи операций с привлечением Hibernate внутри открытого сеанса она становится заметнее. Более существенной мог бы стать макрос для организации выполнения операций в контексте транзакции Hibernate:

```
(defmacro with-transaction
  [& body]
  `(let [~'tx (.beginTransaction ~'session)]    ❶
     ~@body
     (.commit ~'tx)))
```

❶ Так как объекты Transaction в Hibernate поддерживают практичные методы (подобно объектам Session), мы неявно связываем текущий объект Transaction с локальным символом tx, к которому легко можно обратиться из пользовательского кода. Это дополнение делает макрос with-transaction анафорическим.

Здесь session – это имя, которое по нашим ожиданиям должно быть связано с объектом текущего открытого сеанса; то есть мак-

[1] Определяя неявную привязку, видимую пользовательскому коду, макрос with-session автоматически становится *анафорическим* (anaphoric) (иногда такие макросы называют *негигиеничными* (unhygienic)). Дополнительную информацию об анафорических макросах можно найти в разделах «Гигиена» и «Предоставление пользователю права выбора имен», в главе 5.

рос `with-transaction` будет незаметно использовать анафорическое имя `session`, связанное в `with-session`. Сгенерированный код запустит транзакцию, выполнит дополнительные формы в теле и затем подтвердит транзакцию. Это дает нам возможность создавать намного более простые реализации, чем в примере 14.3:

```
(defn add-authors
  [& authors]
  (with-session @session-factory
    (with-transaction
      (doseq [author authors]
        (.save session author)))))
```

Избавившись от шаблонного кода и синтаксических сложностей, связанных с рутиной обслуживания сеансов и транзакций, мы можем сделать свой код, выполняющий операции с базой данных, более удобочитаемым, более простым и, что самое важное, менее чреватый ошибками.

В заключение

В Clojure имеется превосходная поддержка реляционных баз данных. JVM и JDBC образуют мощную основу, а родные для Clojure особенности обеспечивают простые в использовании слои дополнительных функциональных возможностей для создания приложений баз данных. При этом сохраняется возможность вернуться к использованию зрелых и надежных фреймворков для Java, таких как Hibernate.

Глава 15. Нереляционные базы данных

После многих лет, в течение которых реляционные базы данных оставались единственным средством хранения информации, доступным прикладным программистам, появилось множество новых классов баз данных, которые можно расценивать, как достойные альтернативы вездесущим реляционным базам данных. Эти базы данных существенно отличаются друг от друга. Но несмотря на различия, хранилища пар ключ/значение, а также колоночно-ориентированные и документ-ориентированные базы данных сходны в одном – они являются альтернативами традиционной реляционной модели представления данных, доминировавшей долгое время; поэтому их часто упоминают под общим названием *нереляционные базы данных*[1]. Уникальные возможности и увеличивающаяся популярность этих хранилищ данных делают их привычными компонентами в новых приложениях на Clojure, поэтому стоит взглянуть, как выглядит такая комбинация.

CouchDB – документ-ориентированная нереляционная база данных, архитектура которой и используемая модель данных отлично согласуются с особенностями и философией языка Clojure. Их сочетание позволяет упростить реализацию многих типов приложений, от обычных веб-интерфейсов до легко расширяемых систем обмена сообщениями.

Для начала познакомимся с некоторыми особенностями CouchDB, особенно близкими к языку Clojure:

- модель данных определяется исключительно документами JSON, которые легко преобразуются в структуры данных Clojure и обратно;

---

[1] Или под более распространенным (к сожалению) названием «базы данных NoSQL».

- используемая система хранения, основанная на B-деревьях, гарантирует надежность хранения данных и атомарность операций с ними;
- для управления конкурирующими модификациями в многопользовательской среде используется *многовариантный контроль совпадений* (Multiversion Concurrency Control, MVCC) – та же самая оптимистичная модель управления версиями изменений, что применяется в реализации программной транзакционной памяти Clojure;
- позволяет определять сложные запросы в терминах преобразований данных, называемых представлениями, которые могут быть реализованы практически на любом языке, включая Clojure.

Неизменяемость структур данных и четко определенная семантика конкуренции делают естественным сочетание Clojure с базой данных CouchDB, которая использует похожую модель неизменяемых документов и имеет ясную семантику, направленную на сохранность данных, атомарность операций и управление конфликтующими изменениями.

Настройка CouchDB и Clutch

CouchDB – очень зрелая база данных (несмотря на номер 1.2.0 версии, которую мы будем использовать в наших примерах). Огромное количество документации к ней можно найти как в электронном виде на сайте проекта Apache[1], так и в виде книги, выпущенной издательством O'Reilly[2]. Желающие разобраться во всех тонкостях этой базы данных должны обратиться к этим ресурсам.

Прежде чем использовать базу данных CouchDB в программе на Clojure, ее необходимо запустить на локальном компьютере[3]. Во всех наших примерах мы будем использовать Clutch API (https://github.com/ashafa/clutch), библиотеку для Clojure, обеспечивающую

---

[1] http://couchdb.apache.org – (особенно подробную информацию можно найти на вики-странице).

[2] Написанной разработчиками CouchDB и доступной не только в печатном виде, но и бесплатно в электронном виде, по адресу: http://guide.couchdb.org.

[3] Или воспользоваться одним из доступных экземпляров баз данных CouchDB на бесплатном хостинге Cloudant, где можно опробовать все примеры, кроме серверных представлений на Clojure: https://cloudant.com.

полную поддержку возможностей CouchDB. Для этого добавьте библиотеку Clutch в зависимости проекта:

```
[com.ashafa/clutch "0.3.0"]
```

Простейшие CRUD-операции[1]

Приступим к исследованию CouchDB из REPL, начав с самых основ: создания, изменения и удаления документов.

Пример 15.1. Простые взаимодействия с CouchDB в REPL

```
(use '[com.ashafa.clutch :only (create-database with-db put-document
                                get-document delete-document)
                         :as clutch])

(def db (create-database "repl-crud"))                                    ❶

(put-document db {:_id "foo" :some-data "bar"})                           ❷
;= {:_rev "1-2bd2719826", :some-data "bar", :_id "foo"}
(put-document db (assoc *1 :other-data "quux"))                           ❸
;= {:other-data "quux", :_rev "2-9f29b39770", :some-data "bar", :_id "foo"}
(get-document db "foo")                                                   ❹
;= {:_id "foo", :_rev "2-9f29b39770", :other-data "quux", :some-data "bar"}
(delete-document db *1)                                                   ❺
;= {:ok true, :id "foo", :rev "3-3e98dd1028"}
(get-document db "foo")                                                   ❻
;= nil
```

❶ Сначала создадим пустую базу данных для экспериментов в REPL.

❷ Здесь создается документ с применением ассоциативного массива. `put-document` возвращает созданный документ, который представляет собой тот же самый исходный ассоциативный массив, за исключением дополнительного слота `:_rev`. Обратите внимание, что здесь мы определили слот `:_id` – он играет роль «первичного ключа» для документа; если опустить этот слот, CouchDB присвоит новому документу свое значение UUID (Universally Unique Identifier – универсально уникальный идентификатор) в слоте `:_id`.

---

[1] Аббревиатурой CRUD (от англ. Create-Read-Update-Delete) обозначается набор базовых операций: создание, чтение, изменение и удаление. – *Прим. перев.*

❸ Операция изменения документа. Обратите внимание, что значение слота :_rev в возвращаемом значении также изменилось[1].
❹ Простая операция чтения. Она всегда возвращает последнюю версию запрошенного документа. Имеется также возможность запрашивать предыдущие версии документа.
❺ Операция удаления. В данном случае передается ассоциативный массив, содержащий полный документ, однако в действительности достаточно передать только слоты :_id и :_rev со значениями, соответствующими последней версии документа.
❻ Операция чтения возвращает nil, если документ с указанным ключом отсутствует.

Прямые параллели между соответствующими представлениями данных делают взаимодействия между Clojure и CouchDB очень естественными. Для представления данных в CouchDB используется формат JSON (ассоциативные массивы со строковыми ключами, и скалярами, массивами и другими ассоциативными массивами в виде значений), который легко преобразуется в ассоциативные массивы, векторы и скаляры Clojure. К счастью, JSON-парсер, используемый в библиотеке Clutch (https://github.com/clojure/data.json), автоматически преобразует строковые ключи в ассоциативных массивах JSON в ключевые слова Clojure; это позволяет легко находить пары ключ/значение и выполнять обход вложенных структур с использованием функций, таких как get-in, или потоковых макросов ->, ->> и так далее.

Сказанное можно проиллюстрировать примером, показывающим насколько просто извлекать элементы документа, полученного из CouchDB:

```
(clutch/create-document {:_id "foo"
                         :data ["bar" {:details ["bat" false 42]}]})
;= {:_id "foo", :data ["bar" {:details ["bat" false 42]}],
;= :_rev "1-6d7460947434b90bf88f033785f81cdd"}
(->> (get-document db "foo")
  :data
```

[1] Не забывайте, что переменная *1 создается автоматически оболочкой REPL и хранит значение последнего вычисленного выражения. Она напоминает переменную _ в оболочке IRB Ruby или в интерактивной оболочке Python. Подробнее о переменных, создаваемых оболочкой REPL, рассказывается в разделе «Переменные, создаваемые оболочкой REPL», в главе 10.

```
  second
  :details
  (filter number?))
;= (42)
```

Таким образом документы, хранящиеся и извлекаемые из CouchDB, в действительности представляют собой обычные для Clojure структуры данных и к ним могут применяться любые идиоматические для Clojure приемы извлечения и обработки данных. Это значительно упрощает прикладной код, а также позволяет создавать простые и ясные модели данных, что зачастую является одним из наиболее сложных аспектов использования баз данных.

Представления

CouchDB не поддерживает SQL и другие подобные механизмы выполнения запросов. Документы в базе данных могут индексироваться одним лишь «первичным» строковым ключом.

Однако CouchDB поддерживает альтернативный механизм, который называется *представления* (views). Представления очень напоминают концепцию материализованных представлений (materialized views), поддерживаемую некоторыми другими системами реляционных баз данных. Представления:

- хранятся и вызываются отдельно от «исходной» базы данных;
- определяются программно (с использованием практически любого языка, включая Clojure, как будет показано чуть ниже) и заблаговременно;
- отражают текущее, на момент обращения, состояние документов в исходной базе данных.

Основной особенностью представлений в CouchDB, является способность определять практически любые выборки данных и тем самым обеспечивать гибкость, недостижимую для SQL и других специализированных механизмов запросов. С другой стороны, представления CouchDB всегда должны определяться в базе данных заранее, до обращения к ним[1], что обеспечивает чрезвычайно высокую скорость получения результатов, независимо от объема обработки, которую необходимо выполнить, чтобы их воспроизвести.

---

[1] Существует возможность создавать временные представления, но такие представления всегда будут действовать *значительно медленнее* «обычных» представлений, созданных и сохраненных заблаговременно.

Прежде чем приступить к исследованию представлений, заполним новую базу данных с именем *logging* гипотетическими сообщениями. Для этого воспользуемся функцией `bulk-update` из библиотеки Clutch, вызывающей функцию `_bulk_docs` CouchDB API; это самый эффективный способ загрузки больших объемов данных в CouchDB:

```
(clutch/bulk-update (create-database "logging")
  [{:evt-type "auth/new-user" :username "Chas"}
   {:evt-type "auth/new-user" :username "Dave"}
   {:evt-type "sales/purchase" :username "Chas" :products ["widget1"]}
   {:evt-type "sales/purchase" :username "Robin" :products ["widget14"]}
   {:evt-type "sales/RFQ" :username "Robin" :budget 20000}])
```

Простое представление (на JavaScript)

По умолчанию представления CouchDB определяются на языке JavaScript. Для начала познакомимся с приемами обращения к представлениям из Clojure, а затем реализуем пару представлений на самом языке Clojure.

Независимо от языка, используемого для реализации представлений, все они определяются как комбинация функций отображения (map) и свертки (reduce).

> **Внимание.** Понятие «свертки» в CouchDB имеет несколько иной смысл, чем в Clojure и других функциональных языках программирования, а также в других системах обработки данных, таких как фреймворк Hadoop и модель MapReduce пропагандируемая компанией Google. Семантика свертки (и специальное понятие повторной свертки (rereduce) в CouchDB) тесно связана со стратегией индексирования B-деревьев, используемой в CouchDB. За дополнительной информацией обращайтесь по адресу: http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views#Reduce_Functions.

В нашем первом представлении мы реализуем один из типичных случаев их применения: получение информации о количестве имеющихся сообщений каждого типа. Для этого потребуется определить функции отображения и свертки:

```
function(doc) {
     emit(doc["evt-type"], null);
}

function (keys, vals, rereduce) {
     return rereduce ? sum(vals) : vals.length;
}
```

Создать это представление можно с помощью *Futon*, административного интерфейса для CouchDB[1], или с помощью библиотеки Clutch, как показано ниже:

```
(clutch/save-view "logging" "jsviews"          ❶
  (clutch/view-server-fns :javascript          ❷
    {:type-counts                              ❸
     {:map "function(doc) {
              emit(doc['evt-type'], null);
            }"
      :reduce "function (keys, vals, rereduce) {
                 return rereduce ? sum(vals) : vals.length;
               }"}}))
```

❶ `jsviews` определяет имя *дизайн-документа* (design document), где будет храниться представление. Дизайн-документы, это особые документы в базе данных CouchDB, хранящие код реализации представлений, фильтров и других функциональных возможностей, встраиваемых в базу данных.

❷ Данное представление реализовано на языке JavaScript, и это обязательно нужно отметить, чтобы дизайн-документ был настроен правильно.

❸ `type-counts` – это имя представления; в вызове функции `view-server-fns` можно определить любое количество представлений, просто добавляя соответствующие элементы в ассоциативный массив.

Теперь можно обратиться к представлению:

```
(clutch/get-view "logging" "jsviews" :type-counts {:group true})
;= ({:key "auth/new-user", :value 2}
;=  {:key "sales/purchase", :value 2}
;=  {:key "sales/RFQ", :value 1})
```

И получить количество записей с сообщениями каждого типа, определяемого слотом `:evt-type`. Вызов `get-view` возвращает «ленивую» последовательность документов с результатами. Это означает, что мы легко сможем получать и обрабатывать тысячи и даже миллионы документов. И снова, документы с результатами обращения к представлению – это самые обычные ассоциативные массивы и

[1] Если вы используете локальную установку CouchDB, получить доступ к Futon можно с помощью веб-браузера, открыв страницу по адресу: http://localhost:5984/_utils.

значения Clojure, что позволяет с помощью некоторых операций обработки последовательностей преобразовать результаты в более удобную форму:

```
(->> (clutch/get-view "logging" "jsviews" :type-counts {:group true})
  (map (juxt :key :value))
  (into {}))
;= {"auth/new-user" 2, "sales/purchase" 2, "sales/RFQ" 1}
```

Как и следовало ожидать, эти счетчики будут автоматически обновляться по мере записи новых сообщений в базу данных.

> **Примечание.** Параметр `:group` запроса, использованный в примере выше, – это краткая форма свертки значений для каждого уникального ключа. Без этого параметра мы получим единственный результат со значением `5`. Представления обладают самыми разными параметрами запросов, которые мы не использовали здесь; за дополнительной информацией по ним обращайтесь к документации CouchDB (http://wiki.apache.org/couchdb/HTTP_view_API#Querying_Options).

Представления на языке Clojure

Создание представлений для CouchDB на JavaScript – довольно удобная возможность, потому что очень многие знакомы с этим языком, и в состав CouchDB уже входит реализация сервера представлений на JavaScript.

> **Примечание.** Представления CouchDB получаются в результате передачи исходного кода, определяющего множество представлений наряду с данными в формате JSON, соответствующими «исходным» документам, отдельному процессу, который называется «сервером представлений». Сервер представлений просто читает эти данные из `stdin` и выводит результаты обращения к представлению в `stdout`. Такая простота означает, что сервер представлений легко можно реализовать практически на любом языке программирования.
> Вам едва ли придется задумываться о реализации собственного сервера представлений, потому что существуют реализации на десятках разных языков, включая и реализацию на языке Clojure, входящую в состав библиотеки Clutch.

Однако имеются весьма веские доводы в пользу использования сервера представлений на языке, отличном от JavaScript, справедливость которых не вызывает сомнений, когда дело доходит до сервера представлений на языке Clojure, входящего в состав библиотеки Clutch:

- возможность использования богатых возможностей, вероятно, более знакомого языка;
- для выбранного языка сервера представлений почти наверняка имеются высококачественные библиотеки с гораздо большими возможностями, чем доступны в предустановленном сервере представлений на JavaScript;
- если для получения результатов в представлении необходимо производить массивные вычисления, предпочтительнее использовать сервер представлений на языке, обладающем более высокой производительностью, чем JavaScript.

Для настройки локального экземпляра базы данных CouchDB на использование сервера представлений из библиотеки Clutch, обычно требуется вызвать функцию `configure-view-server` в оболочке REPL, где включена поддержка Clutch[1]:

```
(use '[com.ashafa.clutch.view-server :only (view-server-exec-string)])

(clutch/configure-view-server "http://localhost:5984" (view-server-exec-string))
```

Функция `configure-view-server` создаст в экземпляре CouchDB запись о сервере представлений на Clojure, с командой, которая будет вызываться базой данных CouchDB, когда ей потребуется запустить сервер представлений[2].

Для начала перепишем наше простое JavaScript-представление на Clojure. Следующий пример использует Clutch, чтобы сохранить представление в другом дизайн-документе (`clj-views`, вместо `jsviews`, использовавшегося выше для сохранения представления на JavaScript):

```
(clutch/save-view "logging" "clj-views"
  (clutch/view-server-fns :clojure
    {:type-counts
```

[1] Представления в CouchDB, где используется сервер представлений из библиотеки Clutch, можно создавать на языке ClojureScript (см. раздел «ClojureScript» в главе 20) – для этого не придется настраивать новый сервер представлений, что очень удобно, особенно при использовании хостинга Cloudant или другого: https://github.com/clojure-clutch/clutch-clojurescript.

[2] Этот способ настройки предусмотрен только для удобства использования в оболочке REPL; за дополнительной информацией о настройке сервера представлений из библиотеки Clutch в CouchDB обращайтесь к файлу *README* по адресу: https://github.com/clojure-clutch/clutch.

```
      {:map (fn [doc]
              [[(:evt-type doc) nil]])
       :reduce (fn [keys vals rereduce]
                 (if rereduce
                   (reduce + vals)
                   (count vals)))}))
```

Независимо от языка, для доступа ко всем представлениям поддерживается один и тот же прикладной интерфейс, поэтому, кроме имени дизайн-документа, наш код обращения к представлению остался прежним:

```
(->> (clutch/get-view "logging" "clj-views" :type-counts {:group true})
     (map (juxt :key :value))
     (into {}))
;= {["auth/new-user" 2] 2, ["sales/purchase" 2] 2, ["sales/RFQ" 1] 1}
```

Однако наша цель не в том, чтобы заменить квадратные скобки круглыми; создавая представления на Clojure, мы можем пользоваться всеми его возможностями для воспроизведения данных представления, включая библиотеки Clojure и Java.

Взглянув на наш набор данных с сообщениями с предметной точки зрения, можно предположить, что он имеет отношение некоторому приложению электронной коммерции. Похоже, что конкретные типы событий, которые мы видим здесь, образуют некоторую иерархию (например, совершенно очевидно, что типы событий `sales/purchase` и `sales/RFQ` состоят в определенном родстве). Было бы совсем несложно (независимо от языка программирования, на котором написано представление) выделить типы событий по символу слеша и с помощью средств сопоставления, доступных в представлениях CouchDB (http://wiki.apache.org/couchdb/View_collation), выполнить подсчет в группах событий на основе неявной иерархии. В некоторых ситуациях это было бы целесообразно, но такой подход втискивает наши типы событий в строгую иерархию. Решить эту проблему через творческий подход к выбору имен типов или используя некоторый способ определения принадлежности к множеству типов (например, поместив в слот `:evt-type` массив типов) возможно, но не элегантно, не гибко и слишком сложно.

Более удачное решение: определить типы событий в терминах иерархий Clojure[1]; например:

---

[1] Подробнее об иерархиях и мультиметодах в Clojure рассказывается в главе 7.

Пример 15.2. Определение иерархии событий

```
(ns eventing.types)

(derive 'sales/purchase 'sales/all)
(derive 'sales/purchase 'finance/accounts-receivable)
(derive 'finance/accounts-receivable 'finance/all)
(derive 'finance/all 'events/all)
(derive 'sales/all 'events/all)
(derive 'sales/RFQ 'sales/lead-generation)
(derive 'sales/lead-generation 'sales/all)
(derive 'auth/new-user 'sales/lead-generation)
(derive 'auth/new-user 'security/all)
(derive 'security/all 'events/all)
```

Затем эту иерархию можно использовать в нашем сервере представлений для развертывания каждого конкретного типа во все его родительские типы, которые могут вообще никогда не появляться в слоте `:evt-type`, но представляют определенный интерес с предметной точки зрения. Таким способом мы можем обеспечить поддержку типов событий, затрагивающих разные области. Следующее представление на языке Clojure реализует эту стратегию:

Пример 15.3. Представление дополненное иерархией

```
(clutch/save-view "logging" "clj-views"
  (clutch/view-server-fns :clojure
    {:type-counts
     {:map (do
             (require 'eventing.types)                          ❶
             (fn [doc]
               (let [concrete-type (-> doc :evt-type symbol)]   ❷
                 (for [evtsym (cons concrete-type               ❸
                                    (ancestors concrete-type))]
                   [(str evtsym) nil]))))
      :reduce (fn [keys vals rereduce]                          ❹
                (if rereduce
                  (reduce + vals)
                  (count vals)))}}))

(->> (clutch/with-db "logging"
           (clutch/get-view "clj-views" :type-counts {:group true}))
     (map (juxt :key :value))
     (into {}))
```

```
;= {"events/all" 5,
;= "sales/all" 5,
;= "finance/all" 2,
;= "finance/accounts-receivable" 2,
;= "sales/lead-generation" 3,
;= "sales/purchase" 2,
;= "sales/RFQ" 1,
;= "security/all" 2,
;= "auth/new-user" 2}
```

❶ Сначала необходимо обеспечить загрузку пространства имен, определяющего отношения между известными конкретными типами событий и их «предками». Загрузка выполняется только один раз, когда представление материализуется на сервере представлений и перед обработкой любых документов. Обратите внимание, что загружаемое пространство имен (в данном случае `eventing.types`) должно находиться в пути поиска classpath сервера представлений.

❷ Каждая строка `:evt-type` преобразуется в символ, чтобы...

❸ ...можно было получить символы предков из объявленной иерархии. Для каждого символа, включая символ, соответствующий конкретному типу в `:evt-type`, представление выводит один результат.

❹ Функция свертки осталась той же, что и в предыдущей версии представления подсчета сообщений.

Теперь результаты представления выглядят намного интереснее. Определив иерархию типов событий в Clojure, мы получили возможность подсчитывать события не только по конкретным типам, но и по более крупным категориям, что позволяет получить более практичные результаты, чем при использовании лексикографической схемы именования:

- события `auth/new-user` имеют отношение не только к расширению круга потребителей (lead generation), но к обеспечению безопасности системы;
- события `sales/purchase` можно отнести к более широким категориям, таким как расчеты с клиентами и финансы.

Самое замечательное, что наша иерархия типов событий может быть добавлена или реорганизована совершенно независимо от модулей или приложений, которые производят фактические события. Например, если для отчетности потребуется получать сведения о регистрации пользователей наряду с другими данными, родительский тип `security/all` можно объявить потомком нового типа `audit/all`, включив тем самым `auth/new-users` в более широкую категорию без

изменений в системе аутентификации пользователей, генерирующей события `auth/new-user`.

> **Внимание. Функции представлений должны быть чистыми.** При создании представлений важно помнить, что вы не можете управлять моментом вызова представлений или количеством вызовов. Данное предупреждение не относится к представлениям на JavaScript, потому что в этом языке отсутствуют функции ввода/вывода или другие операции с побочными эффектами. Однако при создании представлений на Clojure (или другом языке, отличном от JavaScript), все функции должны быть чистыми[1]. Например, отправка извещения по электронной почте каждый раз, когда представление обрабатывает событие типа `sales/purchase`, может производить разрушительный эффект: электронное письмо будет отправляться всякий раз при изменении документа события покупки или компактификации базы данных, или всякий раз, когда CouchDB решит сделать недействительными результаты представления для данного документа.

_changes: использование CouchDB в роли очереди сообщений

CouchDB поддерживает API извещений об изменениях (с именем `_changes`), дающий клиентам гибко реагировать в ответ на потоки данных.

Вкратце, механизм `_changes` действует следующим образом:

1. Клиент открывает HTTP-соединение с URL `_changes` в базе данных CouchDB.
2. Ожидает. Когда в базе данных произойдут какие-либо изменения (такие как создание, изменение или удаление документа), клиенту будет отправлен ассоциативный массив в формате JSON с идентификатором (ID) и версией измененного документа.
3. Если клиент заинтересован в получении последующих извещений, выполнение продолжается с п. 2.

Самый простой способ использовать механизм `_changes` в Clojure – реализовать вывод всех извещений в `*out*` с помощью функции `watch-changes` из библиотеки Clutch. В следующем примере мы создаем новую базу данных для экспериментов с `_changes`, подключаем к ней функцию-наблюдателя из библиотеки Clutch и добавляем несколько документов, чтобы посмотреть, что произойдет:

---

[1] Обсуждение ссылочной прозрачности и чистых функций можно найти в разделе «Чистые функции», в главе 2.

```
(clutch/create-database "changes")
(clutch/watch-changes "changes" :echo (partial println "changes:"))        ❶

(clutch/bulk-update "changes" [{:_id "doc1"} {:_id "doc2"}])               ❷
;= [{:id "doc1", :rev "5-f36e792166"}
;=  {:id "doc2", :rev "3-5570e8bbb3"}]
; change: {:seq 7, :id doc1, :changes [{:rev 5-f36e792166}]}
; change: {:seq 8, :id doc2, :changes [{:rev 3-5570e8bbb3}]}
(clutch/delete-document "changes" (zipmap [:_id :_rev]
                                    ((juxt :id :rev) (first *1))))
;= {:ok true, :id "doc1", :rev "6-616e3df68"}
; change: {:seq 9, :id doc1, :changes [{:rev 6-616e3df68}], :deleted true}

(clutch/stop-changes "changes" :echo)                                      ❸
;= nil
```

❶ Регистрация новой функции с именем `:echo` в базе данных `changes`; она просто выводит извещения `_changes` в `*out*`.

❷ Теперь мы будем извещены обо всех изменениях в базе данных; эти извещения соответствуют вызовам зарегистрированной нами функции.

❸ `stop-changes` аннулирует подписку на извещения.

Данный механизм является точным аналогом функций-наблюдателей[1], поддерживаемых в Clojure ссылочными типами, такими как атомы, переменные, ссылки и агенты. Концептуально каждая база данных CouchDB содержится внутри отдельного атома; когда в базе данных производятся изменения, они передаются агенту, связанному с вашим вызовом `watch-changes`.

Кроме того, CouchDB позволяет определять функции «фильтров» для программной фильтрации документов, включенных механизмом `_changes` в список измененных документов. эти функции можно определять на любом языке, как и функции-представления, и им можно передавать параметры перед началом приема извещений вызовом `watch-changes`, как будет показано чуть ниже.

Эти механизмы дают почти безграничные возможности для создания очень гибких приложений, управляемых событиями. База данных CouchDB, как очередь сообщений (или, более точно, как механизм, обладающий надмножеством функциональных особен-

[1] Подробнее о функциях-наблюдателях рассказывается в разделе «Функции-наблюдатели», в главе 4.

ностей очередей сообщений) и как хранилище данных (или «система записей», если выражаться терминологией хранилищ данных), обладает различными привлекательными характеристиками, включая:

- отсутствие накладных расходов на синхронизацию или препятствий, мешающих использовать ее как основную базу данных и выделенную очередь сообщений (такую как RabbitMQ, ActiveMQ или одну из различных реализаций JMS);
- простота сопровождения: при прочих равных, одну систему всегда проще поддерживать, чем две.

Внимание. Возможность использовать CouchDB в роли очереди сообщений еще не означает, что вы должны это делать; большая власть подразумевает большую ответственность, и этим все сказано. Стандартные очереди сообщений являются отличным инструментом в своей области, и они замечательно справляются со многими типичными задачами. И все-таки, по нашему мнению опытный практик всегда должен знать о наличии альтернативных решений, которые могут помочь решить уникальную задачу элегантно, или типичную задачу – просто.

Очереди сообщений на заказ

Воспользуемся всеми полученными знаниями об эффективном использовании CouchDB из Clojure для реализации асинхронной очереди для обработки событий, происходящих в базе данных *logging*, рассматривавшейся выше. Для начала вспомним, как выглядят наши данные:

Пример 15.4. Пример информации о событиях

```
{:evt-type "auth/new-user" :username "Chas"}
{:evt-type "auth/new-user" :username "Dave"}
{:evt-type "sales/purchase" :username "Chas" :products ["widget1"]}
{:evt-type "sales/purchase" :username "Robin" :products ["widget14"]}
{:evt-type "sales/RFQ" :username "Robin" :budget 20000}
```

Теперь можно подписаться на получение извещений об изменениях, но вместо того, чтобы просто выводить эти извещения в `*out*`, будем производить некоторую полезную работу; в подобном решении нет ничего неправильного, если ваши требования достаточно просты. Однако в большинстве реальных систем требуется использовать дополнительные рычаги для управления нагрузкой, возникающей при обработке событий, и поддержания модульной архитектуры.

Допустим, что у нас имеется веб-сайт или приложение, генерирующее самые разные события; обработку одних можно отложить на некоторое короткое время, другие могут подождать и дольше, а третьи должны обрабатываться как можно быстрее. Выше приводился пример определения иерархии типов для нашего набора данных. И сейчас мы построим еще одну параллельную иерархию, определяющую различные типы событий по степени срочности их обработки:

Пример 15.5. Иерархия типов событий по степени срочности их обработки

```
(ns eventing.processing)

(derive 'sales/lead-generation 'processing/realtime)
(derive 'sales/purchase 'processing/realtime)

(derive 'security/all 'processing/archive)
(derive 'finance/all 'processing/archive)
```

> **Примечание.** Обратите внимание, что три из четырех объявленных типов связаны не с конкретными типами событий, а с искусственными «категориями» обработки, которые, тем не менее, с успехом можно использовать, благодаря отношениям между конкретными типами и более широкими категориями событий (такими как `'security/all`), установленными выше. В нашем примере этот прием позволяет просто сэкономить на строках кода. Но настоящая выгода появляется, когда несколько команд независимо друг от друга занимаются разработкой отдельных модулей или приложений, каждый из которых генерирует свои события: каждая команда разработчиков может поддерживать собственные иерархии типов (представляющих функциональные, деловые или организационные отношения) без необходимости согласовывать их с теми, кто будет писать модули или приложения для обработки этих событий.

Теперь можно создать фильтр, который позволит отбирать события определенного типа с помощью `isa?`[1].

Пример 15.6. Фильтр, отбирающий события определенного типа

```
(clutch/save-filter "logging" "event-filters"
  (clutch/view-server-fns :clojure
    {:event-isa? (do
                   (require '[eventing types processing])
                   (fn [doc request]
```

[1] Подробнее о семантике `isa?` рассказывается в главе 7.

```
                    (let [req-type (-> request :query :type)
                          evt-type (:evt-type doc)]
                      (and req-type evt-type
                        (isa? (symbol evt-type) (symbol req-type))))))))
```

Это – параметризуемый фильтр (обратите внимание на использование параметра запроса, хранящийся в объекте `request`), поэтому мы можем выбирать, какие события включать в исходящий поток, текущий от механизма `_changes`, опираясь на иерархию, объявленную выше. Прежде чем двинуться дальше, попробуем вывести событие, но с некоторыми дополнениями:

```
(clutch/watch-changes "logging" :echo-leads (partial println "change:")    ❶
  :filter "event-filters/event-isa?"                                        ❷
  :type "sales/lead-generation"                                             ❸
  :include_docs true)                                                       ❹

(clutch/put-document "logging"                                              ❺
  {:evt-type "sales/RFQ" :username "Lilly" :budget 20000})
;= {:_id "8f264da359f887ec3e86c8d34801704b",
;=  :_rev "1-eb10044985c9dccb731bd5f31d0188c6",
;=  :budget 20000, :evt-type "sales/RFQ", :username "Lilly"}
; change: {:seq 26, :id 8f264da359f887ec3e86c8d34801704b,                   ❻
;          :changes [{:rev 1-eb10044985c9dccb731bd5f31d0188c6}],
;          :doc {:_id 8f264da359f887ec3e86c8d34801704b,
;                :_rev 1-eb10044985c9dccb731bd5f31d0188c6,
;                :budget 20000,
;                :evt-type sales/RFQ,
;                :username Lilly}}

(clutch/stop-changes "logging" :echo-leads)
;= nil
```

❶ Как и выше, мы оформляем подписку на события в базе данных *logging*...

❷ ...указываем фильтр, поддерживающий иерархию типов...

❸ ...параметризуем этот фильтр так, чтобы он пропускал только события типа `lead-generation`...

❹ ...и требуем включения полного документа в объект извещения, а не только его слоты `:_id` и `:_rev`.

❺ Когда создается документ с типом в `:evt-type`, являющимся потомком указанного нами типа `lead-generation`...

❻ ...наша функция-наблюдатель примет извещение, как и ожидалось, с полным содержимым документа, соответствующего событию.

Благодаря тому, что мы определили иерархии типов, выражение `(isa? 'sales/RFQ 'sales/lead-generation)` в функции фильтра вернет `true`.

Мы можем определить столько функций-наблюдателей, сколько потребуется для охвата различных служб или уровней приоритетов событий, или просто, чтобы соответствовать некоторым требованиям (особенно если обработка выполняется идемпотентным образом). Каждая функция-наблюдатель играет роль отдельной очереди, по крайней мере, с точки зрения приложения. И снова, благодаря иерархиям, обработка может выполняться произвольным количеством модулей.

Реализуем такую систему. Сначала определим мультиметод[1]:

```
(ns eventing.processing)

(defmulti process-event :evt-type)
```

Затем добавим необходимые реализации `process-event`, в соответствии с иерархиями, объявленными выше. К этому моменту нам будут доступны все средства мультиметода. Наши реализации `process-event` просто выводят некоторый текст с описанием в `*out*`; конечно, настоящие реализации таких методов могли бы делать что-то более существенное: в приложении электронной коммерции они могут отправлять счет, заявку на отправку товара, передавать информацию в систему управления взаимоотношениями с клиентами (Client Relationship Management, CRM) и так далее.

Пример 15.7. Реализация «срочной» обработки событий, связанных с продажами

```
(ns salesorg.event-handling
  (use [eventing.processing :only (process-event)]))

(defmethod process-event 'sales/purchase
  [evt]
  (println (format "We made a sale of %s to %s!" (:products evt)
                                                 (:username evt))))
(defmethod process-event 'sales/lead-generation
  [evt]
  (println "Add prospect to CRM system: " evt))
```

Наконец, зарегистрируем функцию-наблюдателя для обработки каждого события, которая после выемки фактического документа, вы-

[1] См. главу 7.

звавшего событие, удалит из него слоты :_id и :_rev, добавленные базой данных CouchDB и преобразует строку :evt-type с именем конкретного типа в символ, чтобы обеспечить диспетчеризацию события внутри мультиметода process-event не на основе строки, возвращаемой базой данных, а на основе сконструированной нами иерархии. Мы сделаем все это в оболочке REPL, а затем снова воссоздадим пять событий:

```
(require 'eventing.processing 'salesorg.event-handling)

(clutch/watch-changes "logging" :process-events
  #(-> %
      :doc
      (dissoc :_id :_rev)
      (update-in [:evt-type] symbol)
      eventing.processing/process-event)
  :filter "event-filters/event-isa?"
  :type "processing/realtime"
  :include_docs true)

(clutch/bulk-update "logging"
  [{:evt-type "auth/new-user" :username "Chas"}
   {:evt-type "auth/new-user" :username "Dave"}
   {:evt-type "sales/purchase" :username "Chas" :products ["widget1"]}
   {:evt-type "sales/purchase" :username "Robin" :products ["widget14"]}
   {:evt-type "sales/RFQ" :username "Robin" :budget 20000}])
; Add prospect to CRM system: {:evt-type auth/new-user, :username Chas}
; Add prospect to CRM system: {:evt-type auth/new-user, :username Dave}
; We made a sale of ["widget1"] to Chas!
; We made a sale of ["widget14"] to Robin!
; Add prospect to CRM system: {:budget 20000, :evt-type sales/RFQ,
;                              :username Robin}
```

В заключение

Оба инструмента, Clojure и CouchDB, хорошо подходят для работы с разнородными, слабоструктурированными наборами данных – определяющая характеристика для динамического процесса разработки, основанного на моделировании, а также часто спасение для приложений, которые должны интегрироваться в устаревшие и неподдающиеся изменениям системы. Вместе они – мощная комбинация, учитывая, как много дает Clojure в смысле увеличения практической ценности представлений и фильтров CouchDB, а также поддерживает средства, позволяющие моделировать, расширять и максимально использовать особенности CouchDB.

Глава 16. Clojure и Веб

Веб-разработку можно смело считать одним из самых обширных направлений: современные программисты, за редким исключением, не только знают, как создавать веб-приложения, но и регулярно занимаются этим. Коль скоро это так, любой язык общего назначения должен предлагать максимально удобные средства и возможности для создания веб-приложений. Clojure с успехом оправдывает эти ожидания.

Учитывая мощную поддержку JVM и наличие удобных механизмов взаимодействий с Java, при использовании языка Clojure не приходится начинать на пустом месте, с программирования сокетов или с создания модуля для Apache: к вашим услугам вся веб-инфраструктура Java, проверенная и надежная, которая только и ждет, чтобы ее задействовали. В то же время в экосистеме Clojure были выращены и развиты собственные идиомы и принципы проектирования веб-приложений, которые существенно отличаются от общепринятой практики в Java.

«Стек Clojure»

Мы постоянно повторяем как мантру принципы хорошего проектирования приложений на языке Clojure: преимущество обобщенных абстракций перед конкретными типами и реализациями, чистых функций с неизменяемыми данными перед методами с побочными эффектами и изменяемым состоянием, и гибкое объединение этих основных принципов в композиции, которые сами являются надежными строительными блоками. В свете этого уже не кажется удивительным отсутствие специализированного «стека Clojure», по крайней мере на фоне монолитных фреймворков, которые в других языках часто позиционируют как «веб-стеки». Вместо этого сообществом Clojure было создано множество модульных библиотек, которые в совокупности способны удовлетворить все требования веб-разработчика, но они основаны на базовых абстракциях Clojure

и приемах функционального программирования. Вы и ваши коллеги можете использовать их для создания собственного стека, наиболее полно отвечающего потребностям вашим и вашего приложения, предметной области, а также личным приемам и навыкам[1].

Эта философия может показаться нездоровой многим опытным разработчикам, использующим Rails, Django или Lift. Характерные особенности таких «полных стеков» являются следствием следующих причин: люди стремятся повторно реализовать что-то, если не находят необходимых им возможностей. Однако, учитывая контекст, в котором действуют подобные фреймворки, – полноценные объектно-ориентированные модели с явными контроллерами и представлениями – в распоряжении разработчиков оказывается не так много обобщенных абстракций, поэтому создание небольших, специализированных модулей оказывается почти невозможным делом. Учитывая, насколько просто это дается на языке Clojure, подобные монументальные фреймворки не дают той экономии трудозатрат, как в других языках. Проще говоря, если вы привыкли к полным стекам, попытайтесь абстрагироваться от них, пока не попробуете создать пару приложений; мы полагаем, что желания использовать их у вас поубавится.

Мы будем обсуждать веб-приложение, состоящее из трех частей[2], в каждой из которых будет рассматриваться отдельная библиотека (выбранная из нескольких альтернатив):

- обработка запросов и ответов – эта работа выполняется либо HTTP-сервером, либо приложением, подключенным к нему, и заключается в конструировании объектов ответов, соответствующих входящим запросам и выводе HTTP-ответов с соблюдением всех формальностей;
- маршрутизация запросов – порядок передачи запросов по цепочке до вашего обработчика;

---

[1] Как вариант, можете попробовать использовать один из новейших веб-фреймворков, построенных по принципу «все включено», о которых упоминается в конце этой главы.

[2] Таких частей может быть больше или меньше, в зависимости от требований. Они могут решать вопросы, связанные с аутентификацией, проверкой форм, обменом информацией о поддерживаемых типах содержимого (content negotiation) и т. д. и т. п.. Для удовлетворения всех этих требований конечно же существуют отдельные библиотеки на Clojure, но чтобы охватить весь комплекс задач веб-разработки потребовалось бы написать отдельную книгу. Поэтому мы покажем вам лишь начало правильного пути.

- работа с шаблонами – порядок преобразования ответов, возвращаемых обработчиком, в разметку HTML (или любой другой желаемый формат).

В этой главе мы хотели бы показать вам комбинацию из особенно популярных библиотек:

- Ring – основной инструмент обработки запросов и ответов;
- Compojure – реализует маршрутизацию;
- Enlive – обеспечивает поддержку шаблонов.

Другие отдают предпочтение другим вариантам, лучше отвечающим их требованиям. Moustache[1] – еще один замечательный механизм маршрутизации. Часто для работы с шаблонами выбирается библиотека Hiccup[2], clostache[3] – механизм управления шаблонами в стиле Mustache[4], доступны для многих для многих других языков, и даже имеется возможность использовать JSP, Velocity, stringtemplate и другие библиотеки из веб-пространства Java.

Если после демонстрации совокупности библиотек в этой главе у вас появится желание познакомиться с другими альтернативами, отличной отправной точкой для вас может послужить http://brehaut.net/blog/2011/ring_introduction.

Основа: Ring

Как демонстрировалось на протяжении всей книги, Clojure превосходно справляется с задачами преобразования данных. Однако гораздо лучше, когда вообще нет необходимости преобразовывать данные, то есть, когда для решения задач в некоторой предметной области уже имеется подходящий формат представления данных, и нет нужды изобретать что-то новое. В духе и под некоторым влиянием Python WSGI и Ruby Rack, библиотека Ring[5] определяет стандартную схему представления веб-запросов и ответов в виде структур данных на языке Clojure и несколько ключевых струк-

---

[1] https://github.com/cgrand/moustache.

[2] Доступна по адресу: https://github.com/weavejester/hiccup. Чтобы получить некоторое представление о Hiccup, загляните еще раз в раздел «Расширение HTML DSL» в главе 13, где мы реализовали некоторое подмножество функций этой библиотеки с целью поупражняться в тестировании.

[3] https://github.com/fhd/clostache.

[4] http://mustache.github.com/.

[5] https://github.com/ring-clojure/ring.

турных концепций, основанных на композиции функций: *адаптеры* (adapters), *обработчики* (handlers) и *промежуточные функции* (middleware).

Понимание спецификации Ring SPEC[1] крайне важно для эффективной разработки веб-приложений на языке Clojure. Здесь мы исследуем каждый из аспектов, и приведем дословно некоторые фрагменты спецификации[2]. Мы рекомендуем прочитать эту спецификацию целиком хотя бы один раз и постоянно держать ее текст поблизости, пока вы будете учиться работать с данными и абстракциями, которые она определяет.

Запросы и ответы

В отличие от многих других фреймворков, определяющих один фиксированный API для доступа к данным в веб-запросах (таким как запрашиваемый URI, заголовки запроса, параметры запроса, содержимое, и так далее) и другой API – для отправки ответов, библиотека Ring представляет запросы и ответы в виде обычных ассоциативных массивов. В обоих случаях эти ассоциативные массивы обязаны содержать определенные слоты, могут содержать некоторые другие слоты и использоваться для хранения любых посторонних данных, которые вам могут потребоваться в процессе обработки.

Список ключей в ассоциативных массивах для запросов в библиотеке Ring перечислены в табл. 16.1 (необязательные слоты обозначены *наклонным шрифтом*).

Таблица 16.1. Ассоциативные массивы для запросов в Ring

Ключ	Описание значения
`:server-port`	Порт, на который поступил обрабатываемый запрос
`:server-name`	Имя сервера или IP-адрес в виде строки
`:remote-addr`	IP-адрес клиента или последнего прокси-сервера, отправившего запрос
`:uri`	URI запроса в виде строки. Должен начинаться с символа "/"
`:scheme`	Транспортный протокол, должен иметь значение `:http` или `:https`

[1] Опубликована по адресу: https://github.com/mmcgrana/ring/blob/master/SPEC.

[2] Библиотека Ring и ее спецификация распространяются на условиях лицензии MIT, Copyright © 2009–2010 Mark McGranaghan.

Таблица 16.1. Ассоциативные массивы для запросов в Ring

Ключ	Описание значения
`:request-method`	HTTP-метод запроса, должен иметь значение `:get`, `:head`, `:options`, `:put`, `:post` или `:delete`
`:headers`	Ассоциативный массив с именами заголовков в нижнем регистре и строковыми значениями
`:content-type`	MIME-тип тела запроса в виде строки, если известен
`:content-length`	Длина тела запроса в байтах, если известна
`:character-encoding`	Имя кодировки символов, составляющих строку с телом запроса, если известно
`:query-string`	Строка запроса, если имеется
`:body`	Экземпляр `java.io.InputStream` с телом запроса, если имеется

Эта схема включает основные данные, связанные с единственным HTTP-запросом. Например, допустим была выполнена попытка обратиться по адресу: https://company.com:8080/accounts?q=Acme; тогда соответствующий ассоциативный массив с запросом в библиотеке Ring выглядел бы примерно так:

```
{:remote-addr "127.0.0.1",
 :scheme :http,
 :request-method :get,
 :query-string "q=Acme",
 :content-type nil,
 :uri "/accounts",
 :server-name "company.com",
 :content-length nil,
 :server-port 8080,
 :body #<ByteArrayInputStream java.io.ByteArrayInputStream@604fd0e9>,     ❶
 :headers
 {"user-agent" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6) Firefox/8.0.1",
  "accept-charset" "ISO-8859-1,utf-8;q=0.7,*;q=0.7",
 "accept" "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "accept-encoding" "gzip, deflate",
  "accept-language" "en-us,en;q=0.5",
  "connection" "keep-alive"}}
```

❶ Так как это простой GET-запрос, экземпляр `:body InputStream` будет пуст; и оба слота, `:content-type` и `:content-length`, отсутствуют в ассоциативном массиве.

Это – обычный ассоциативный массив, и может обрабатываться точно так же, как любые другие ассоциативные массивы.

Ответы в библиотеке Ring точно так же представлены ассоциативными массивами (табл. 16.2). Два слота в них являются обязательными и один (`:body`) необязательный.

Таблица 16.2. Ассоциативные массивы для ответов в Ring

Ключ	Описание значения
`:status`	Код HTTP-состояния, значение должно быть больше или равно 100
`:headers`	Ассоциативный массив с именами и значениями заголовков. Значениями в нем могут быть строки, тогда заголовок будет послан в HTTP-ответе в виде пары имя/значение, или последовательностью строк, и тогда для каждого значения будет послана отдельная пара имя/значение
`:body`	Необязательная строка, последовательность строк, экземпляр `java.io.File` или `java.io.InputStream`

Возможно теперь вы начинаете представлять порядок обработки веб-запросов с помощью Ring: ваш код будет выполнять некоторые операции, соответствующие ассоциативному массиву запроса, и возвращать ассоциативный массив ответа. Например, предыдущий GET-запрос можно с уверенностью считать запросом на получение HTML-страницы. Ответ на него мог бы иметь следующий вид:

```
{:status 200
 :headers {"Content-Type" "text/html"}
 :body "<html>...</html>"}
```

При желании в слоте `:body` ответа можно передавать значения других типов; если в ответ нужно отправить статический файл, хранящийся на диске, можно передать его явно:

```
{:status 200
 :headers {"Content-Type" "image/png"}
 :body (java.io.File. "/path/to/file.png")}
```

Наконец, как будет показано ниже, библиотека полностью совместима со служебными вызовами HTTP API, не требующими определять тело ответа. Например, в ответ на HTTP-запрос PUT выгрузки файла можно было бы сообщить, что запрос принят и на

стороне сервера создан соответствующий ресурс, просто отправив HTTP-код состояния 201:

```
{:status 201 :headers {}}
```

К данному моменту у вас, возможно, уже зародился вопрос – как получаются ассоциативные массивы, представляющие запросы, и как наш веб-сервер превращает ассоциативные массивы ответов в соответствующие HTTP-ответы. Эта функция возлагается на адаптеры.

Адаптеры

Адаптер из библиотеки Ring – это мост между приложением и реализацией протокола HTTP и/или сервера. Проще говоря, когда принимается HTTP-запрос, адаптер преобразует его в ассоциативный массив запроса и передает приложению для обработки. В ответ приложение должно вернуть адаптеру ассоциативный массив ответа, который будет использоваться адаптером для отправки HTTP-ответа клиенту.

Вам едва ли когда-нибудь придется писать собственные адаптеры, но знать, как они вписываются в общую структуру Ring, очень важно. Существует множество реализаций адаптеров, позволяющих приложениям на основе библиотеки Ring взаимодействовать с различными HTTP-серверами и HTTP API:

Сервлеты. Библиотека Ring включает адаптер, позволяющий приложениям на основе Ring выступать в роли Java-сервлетов, пригодных для развертывания на любых серверах веб-приложений на Java. Эта тема более подробно обсуждается в разделе «Упаковка веб-приложений», в главе 17.

ring-jetty-adapter. В состав библиотеки Ring входит также адаптер `ring-jetty-adapter`, используемый для обслуживания запросов под управлением встраиваемого HTTP-сервера Jetty (http://jetty.codehaus.org/jetty/). Это наиболее типичный способ выполнения Ring-приложений, и мы познакомимся с ним чуть ниже.

ring-httpcore-adapter. Этот адаптер (https://github.com/mmcgrana/ring-httpcore-adapter) близко напоминает адаптер `ring-jetty-adapter`, но используется при работе под управлением встраиваемого сервера Apache HTTPCore.

Aleph. Если у вас сложилось ощущение, что библиотека Ring является довольно легковесной, то вы не ошиблись. Фактически, ядро

библиотеки Ring не является реализацией какого-то конкретного механизма (и действительно имеет очень небольшой размер); оно лишь определяет схему данных запрос/ответ и ключевые понятия *адаптеров*, *промежуточных функций* (middleware) и *обработчиков*. Благодаря этим абстракциям, библиотека Ring – каноническая реализация которой имеет синхронную природу, соответствующую синхронной природе большинства веб-приложений – может замещаться другими Ring-совместимыми реализациями. Известным примером является Aleph[1] – Ring-совместимый адаптер, использующий Netty[2] для асинхронного обслуживания ответов и не требующий изменений в Ring-приложениях для его использования.

Существуют и другие адаптеры, позволяющие связывать Ring-приложения, например, с серверами Mongrel и FastCGI.

Теперь можно перейти к части, которая выполняет «фактическую» работу в Ring-приложениях – обработчикам – и посмотреть, как объединять их с адаптерами при создании веб-приложений.

Обработчики

Обработчик в терминологии Ring – это обычная функция, принимающая ассоциативный массив с запросом и возвращающая ассоциативный массив с ответом. Все Ring-приложения состоят из множества функций-обработчиков, объединяемых и комбинируемых для достижения необходимого поведения и функциональности.

Начнем с простого эхо-сервера (echo server). Для начала добавим в проект зависимость от Ring[3]:

```
[ring "1.0.0"]
```

Теперь запустим оболочку REPL и напишем веб-приложение:

Пример 16.1. Запуск Ring-приложения из REPL

```
(use '[ring.adapter.jetty :only (run-jetty)])
;= nil
```

[1] https://github.com/ztellman/aleph.

[2] http://www.jboss.org/netty.

[3] Не пугайтесь обилием номеров версий «1.0.0». Все используемые здесь проекты широко используются уже на протяжении многих лет; недавно многим из них одновременно был присвоен номер версии «1.0.0», чтобы засвидетельствовать их стабильность.

```
(defn app                                                    ❶
  [{:keys [uri]}]
  {:body (format "You requested %s" uri)})
;= #'user/app
(def server (run-jetty #'app {:port 8080 :join? false}))     ❷
;= #'user/server                                            ❸
```

❶ Это – функция-обработчик. Все обработчики принимают единственный аргумент – ассоциативный массив запроса, и должны вернуть ассоциативный массив ответа. Для начала мы просто вернем URI запроса в виде простого текста.

❷ Мы используем адаптер для сервера Jetty. Все адаптеры реализуются как функции, принимающие два аргумента: функцию-обработчик для обслуживания запросов, и ассоциативный массив с параметрами настройки адаптера. Здесь мы определяем, что сервер Jetty будет прослушивать порт `8080`, и указываем, что поток выполнения, используемый сервером Jetty, не должен «присоединяться» к приложению; если этого не сделать, оболочка REPL будет блокироваться в ожидании завершения работы сервера Jetty.

❸ Мы решили сохранить ссылку на сервер Jetty в переменной `server`. Это даст нам возможность (если потребуется) остановить сервер вызовом (`.stop server`).

Теперь можно обратиться к запущенному веб-приложению (рис. 16.1).

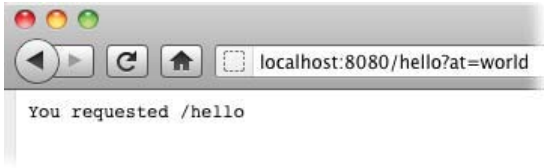

Рис. 16.1. Результат обращения к веб-приложению из браузера

Отлично! Но, обратите внимание, что URI не включает параметры запроса, `?at=world`. Поскольку мы передали адаптеру Jetty переменную с функцией-обработчиком (то есть, `#'app`), а не саму функцию, мы легко можем переопределить функцию, не перезапуская Jetty:

```
(defn app
  [{:keys [uri query-string]}]
  {:body (format "You requested %s with query %s" uri query-string)})
;= #'user/app
```

Результат можно наблюдать немедленно (рис. 16.2).

Рис. 16.2. Результат обращения к измененному веб-приложению

Разумеется, мы можем добиться большего. Было бы крайне неудобно вручную извлекать отдельные параметры запроса из строки. Однако в описании ассоциативных массивов запросов (в спецификации Ring) ничего не говорится о запросах и о возможности получения параметров в каком-то другом виде.

К счастью, это не является большой проблемой. Эта типичная задача легко решается с помощью промежуточных функций.

Промежуточные функции

Промежуточные функции (middleware) – это инструмент расширения или изменения эффекта, производимого обработчиками. Не забывайте, что запросы и ответы представлены ассоциативными массивами, и они легко могут трансформироваться. А так как обработчики – это всего лишь функции, мы легко можем составлять композиции из различных функций, чтобы добиться нужного поведения. Обычно желаемый эффект достигается за счет создания промежуточной функции, как функции высшего порядка, принимающей один или более обработчиков (возможно, с дополнительными параметрами настройки), и возвращающей новый обработчик с требуемой функциональностью.

Воплотим это на практике. Как только что выяснилось, запросы Ring по умолчанию не включают параметры запроса в каком-либо структурированном виде. Мы можем исправить этот недостаток, добавив в приложение промежуточную функцию; эта функция будет добавлять в ассоциативный массив запроса, полученный от адаптера Jetty, другой ассоциативный массив, с параметрами запроса:

```
(use '[ring.middleware.params :only (wrap-params)])
;= nil
```

```
(defn app*                                                          ❶
  [{:keys [uri params]}]                                            ❷
  {:body (format "You requested %s with query %s" uri params)})
;= #'user/app*
(def app (wrap-params app*))                                        ❸
;= #'user/app
```

❶ Теперь мы определили наш обработчик, как `app*`, поэтому обработчик, расширенный промежуточной функцией, можно сохранить в переменной `app`, используемой адаптером Jetty.

❷ Теперь наш обработчик извлекает параметры запроса не из строки `:query-string` в ассоциативном массиве запроса, а из слота `:params`.

❸ Объединение обработчика с промежуточной функцией выполняется обычным вызовом функции высшего порядка. Новый обработчик, возвращаемый функцией `wrap-params`, теперь образует самый внешний слой нашего приложения – после выполнения своей работы, связанной с разбором параметров в строке запроса и в теле POST-запроса (если необходимо), она вызовет функцию-обработчик и передаст ей ассоциативный массив запроса, с вновь созданным слотом `:params`, содержащим параметры из строки запроса.

Взгляните на рис. 16.3.

Рис. 16.3. Результат работы промежуточной функции

Потрясающе, теперь у нас есть параметры, на основе которых мы можем управлять поведением приложения.

Библиотека включает множество готовых промежуточных функций, для включения в веб-приложения, от парсинга cookies и сеансовых данных из заголовков запросов, до поддержки получения форм, состоящих из нескольких частей и операций выгрузки файлов[1]. Сама библиотека Ring по умолчанию не применяет никакие

[1] Полный список промежуточных функций в библиотеке Ring можно найти на главной странице проекта: https://github.com/ring-clojure/ring.

промежуточные функции к обработчикам, однако некоторые другие веб-фреймворки на ее основе делают это.

Наконец, так как промежуточные функции в действительности являются формой композиции функций, создание новой промежуточной функции не представляет сложностей. Пример простой промежуточной функции приводится в разделе «Ring», в главе 12. Благодаря простоте реализации и значительной гибкости, многие расширения для Ring реализуются в терминах промежуточных функций.

Примечание. Концептуально промежуточные функции в библиотеке Ring напоминают сервлет-фильтры в Java: в обоих случаях предусматривается возможность пост-обработки веб-запросов и ответов. С другой стороны, промежуточные функции значительно проще в реализации и использовании – достаточно определить и затем вызвать функцию высшего порядка – в сравнении с сервлет-фильтрами, разработка которых напоминает блуждание по лабиринту различных интерфейсов, а после их развертывания требуется выполнять дополнительные настройки. Но самое досадное в том, что сервлет-фильтры не могут эффективно комбинироваться из-за конфликтов реализаций типов `ServletRequest` и `ServletResponse`, и, если сервлет императивно отправляет содержимое наружу, минуя `ServletResponse`, фильтр не способен перехватить его. Здесь промежуточные функции проявляются во всем своем блеске: благодаря преимуществу модели Ring, где запросы и ответы являются неизменяемыми коллекциями, подчиненными единственной общей абстракции, промежуточные функции могут объединяться в эффективные агрегаты без всяких ограничений... в конце концов, они – всего лишь обычные функции!

Маршрутизация запросов с помощью Compojure

Пока мы определили единственную функцию, обрабатывающую все запросы, однако, кроме самых простых случаев, таких функций будет больше[1]. Нам необходима возможность структурировать приложение более естественным образом, выделяя различные логические функции в разные обработчики, возможно находящиеся в разных пространствах имен, и вызывать их для обработки соответствующих запросов. Можно было бы анализировать строки URI и делегировать обработку запросов другим функциям, однако есть

[1] Конечно, для очень маленькой веб-службы может оказаться вполне достаточно единственного обработчика, запускаемого адаптером Jetty.

более удачное решение, аналогичное тому, как мы объединяли наши обработчики с промежуточными функциями.

Выражаясь простым языком, *маршрутизация* – это процедура выбора обработчика для ответа на веб-запрос, а *маршруты* – это шаблоны атрибутов запроса, используемые для управления процедурой выбора. Абстрактно говоря, веб-приложение можно представить в виде таблицы маршрутов с соответствующими обработчиками, объявленными в разных пространствах имен (рис. 16.4).

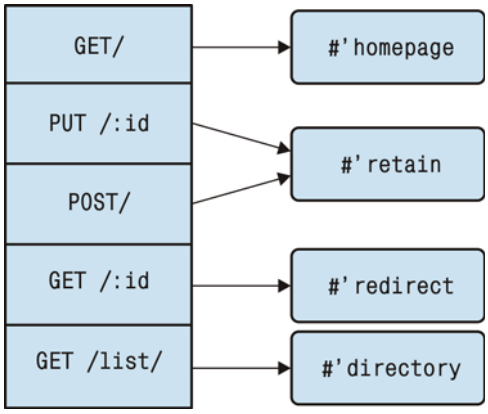

Рис. 16.4. Таблица маршрутов с обработчиками

Когда будет принят GET-запрос на корневой адрес (/) в приложении, он будет передан обработчику `homepage`. Когда будет принят PUT-запрос на любой URI с единственным сегментом (представлен как `:id` на диаграмме), он будет передан обработчику `retain`; то же относится к POST-запросам на корневой адрес в приложении, и так далее.

Попробуем создать законченное веб-приложение, реализующее схему маршрутизации, представленную на рис. 16.4. В результате должна получиться простая HTTP-служба сокращения адресов URL[1], напоминающая bit.ly, tinyurl.com и другие. Такие службы по-

[1] Кто-то мог бы назвать ее *REST*-службой, но семантика REST для многих остается непонятной, как отмечает Рой Филдинг (Roy Fielding) (изобретатель термина REST), например, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. Но мы скажем, что наше приложение – это обычная HTTP-служба.

зволяют пользователям определять и использовать короткие URL, обеспечивающие перенаправление по их каноническим адресам. Для этого воспользуемся библиотекой Compojure (https://github.com/weavejester/compojure) – одним из наиболее популярных инструментов определения *маршрутов* в Ring-приложениях, берущим пример с механизмов маршрутизации во многих фреймворках, которые могут быть вам знакомы, включая Ruby on Rails и Django.

Запустите новый сеанс REPL, предварительно добавив Compojure в список зависимостей проекта, который теперь должен содержать обе библиотеки, Compojure и Ring:

```
[compojure "1.0.1"]
[ring "1.0.1"]
```

Сначала нам нужно определиться, какую модель использовать для хранения URL и их коротких идентификаторов. Мы должны сосредоточиться на специфике создания веб-приложений, поэтому, не мудрствуя лукаво, просто будем хранить текущее состояние в ассоциативном массиве, в памяти, используя ссылочный тип[1].

Какие ссылочные типы было бы предпочтительнее использовать? Мы собираемся дать пользователям возможность присваивать короткие идентификаторы адресам URL, поэтому нам следует защититься от конфликтов при попытках повторного использования уже занятых идентификаторов, и нам требуется координировать изменения в ассоциативном массиве, чтобы параллельные запросы не могли одновременно добавить элементы с идентичными ключами. Чтобы соответствовать поставленным условиям, ассоциативный массив должен храниться в ссылке: каждое изменение в ассоциативном массиве будет выполняться в рамках транзакции, поэтому мы сможем избежать затирания уже зарегистрированных идентификаторов, и параллельные запросы (например, когда два клиента пытаются одновременно зарегистрировать свои URL с одним и тем же коротким идентификатором) никогда не смогут привести нашу модель в противоречивое состояние.

С другой стороны, такие строгие гарантии не нужны для идентификаторов, генерируемых автоматически, когда пользователи не

[1] Если бы мы собирались создать службу для фактического использования, мы могли бы использовать одну из баз данных, обсуждавшихся в главах 14 и 15.

указывают свой собственный идентификатор. Для этой цели можно было бы использовать некоторый алгоритм хеширования или генерировать случайные идентификаторы, но самый простой путь – использовать наращиваемый счетчик. Для управления таким счетчиком лучше всего подходит атом[1].

Итак, мы решили хранить модель в памяти, счетчик в атоме и ассоциативный массив в ссылке:

```
(def ^:private counter (atom 0))

(def ^:private mappings (ref {}))
```

Теперь необходимо определить пару специальных функций для работы с этим состоянием. Мало того, что они упрощают тестирование базовой функциональности вне веб-контекста, если бы нам пришлось перейти на использование базы данных, потребовалось бы изменить только эти две функции:

```
(defn url-for
  [id]
  (@mappings id))

(defn shorten!
  "Сохраняет указанный URL под новым или под указанным идентификатором.
   Возвращает идентификатор в виде строки. Изменяет соответственно
   глобальный ассоциативный массив."
  ([url]
    (let [id (swap! counter inc)          ❶
          id (Long/toString id 36)]       ❷
      (or (shorten! url id)               ❸
          (recur url))))
  ([url id]
    (dosync
      (when-not (@mappings id)            ❹
        (alter mappings assoc id url)
        id))))
```

[1] Так как изменения состояния ссылки могут привести к повторениям из-за конфликтов между запросами, некоторые автоматически сгенерированные идентификаторы, получаемые из счетчика, будут отброшены. В этом нет ничего необычного, но не следует забывать об этом.

❶ swap! вернет новое значение счетчика в атоме, которое гарантированно будет уникальным для данного вызова shorten!.
❷ Вызов Long/toString преобразует числовой идентификатор (ID) в строковое представление числа в системе счисления с основанием 36, которое гораздо компактнее для больших значений, чем строковое представление числа в системе счисления с основанием 10.
❸ Мы повторно используем другую реализацию (арность) функции shorten!, чтобы попытаться сохранить связь сгенерированного идентификатора с URL. Если эта попытка потерпит неудачу (из-за наличия ID в ассоциативном массиве), мы просто рекурсивно повторяем попытку.
❹ Если новый идентификатор (ID) не был добавлен ранее, вызывается assoc, чтобы связать его с исходным URL в ассоциативном массиве, и возвращается короткий идентификатор, в противном случае возвращается nil.

Посмотрим, как работает наша модель:

```
(shorten! "http://clojurebook.com")
;= "1"
(shorten! "http://clojure.org" "clj")
;= "clj"
(shorten! "http://id-already-exists.com" "clj")
;= nil
@mappings
;= {"clj" "http://clojure.org", "1" "http://clojurebook.com"}
```

Пока все хорошо. Теперь определим пару функций для обработки HTTP-запросов ожидаемых типов. Функция retain будет служить «веб-интерфейсом» для shorten!; она делегирует выполнение операций с ассоциативным массивом функции shorten! и возвращает ассоциативный массив ответа, соответствующий результатам попытки сохранить исходный URL (и, возможно, указанный идентификатор ID):

```
(defn retain
  [& [url id :as args]]
  (if-let [id (apply shorten! args)]
    {:status 201                                                    ❶
     :headers {"Location" id}
     :body (list "URL " url " assigned the short identifier " id)}  ❷
    {:status 409 :body (format "Short URL %s is already taken" id)})) ❸
```

❶ Если удалось сохранить URL с конкретным идентификатором (ID), функция `retain` вернет ассоциативный массив ответа со слотом `:status` и значением `201` (HTTP-код ответа, указывающий, что ресурс был успешно создан), с заголовком `Location`, содержащим сохраненный идентификатор ID (который может пригодиться HTTP-клиентам, не являющимся браузерами), и слотом `:body`, содержащим текст для человека.

❷ Как отмечалось в табл. 16.2, тело ответа в ассоциативном массиве может быть значением разных типов, включая последовательность строк.

❸ Единственная причина, не позволяющая сохранить URL с коротким идентификатором, – если запрошенный пользователем идентификатор уже занят. В этом случае возвращается ответ с кодом 409 (HTTP-код ответа, указывающий на «конфликт» препятствующий выполнению операции).

Следующая функция `redirect` ищет запрошенный пользователем идентификатор и в случае успеха посылает браузеру ответ, переадресующий его на найденный URL:

```
(require 'ring.util.response)

(defn redirect
  [id]
  (if-let [url (url-for id)]
    (ring.util.response/redirect url)
    {:status 404 :body (str "No such short URL: " id)}))
```

Если указанное значение `id` будет найдено в ассоциативном массиве `mappings`, `redirect` вызовет вспомогательную функцию из пространства имен `ring.util.response` в библиотеке Ring[1], чтобы послать соответствующий ответ (HTTP-код состояния 302).

Формально функции `retain` и `redirect` пока не являются обработчиками Ring: даже при том, что они возвращают ответы в формате Ring, они не принимают ассоциативный массив запроса. Хоть это и не требовалось, но мы постарались ограничить область применения этих функций, чтобы сделать их более простыми, краткими и несложными в тестировании.

---

[1] Пространство имен `ring.util.response` содержит множество удобных вспомогательных функций для создания и изменения ассоциативных массивов ответов.

Наконец, можно определить маршруты в стиле Compojure, в том же порядке, в каком они следуют на рис. 16.4[1]:

Пример 16.2. Маршруты Compojure для службы сокращения адресов URL

```
(use '[compojure.core :only (GET PUT POST defroutes)])
(require 'compojure.route)

(defroutes app*
  (GET  "/" request "Welcome!")
  (PUT  "/:id" [id url] (retain url id))
  (POST "/" [url] (retain url))
  (GET  "/:id" [id] (redirect id))
  (GET  "/list/" [] (interpose "\n" (keys @mappings)))
  (compojure.route/not-found "Sorry, there's nothing here."))
```

Форма `defroutes` определяет переменную (в данном случае `app*`), содержащую единственный обработчик Ring, который пытается передать запрос каждому обработчику из перечисленных в теле формы в указанном порядке. На первом же обработчике, вернувшем непустое значение, процесс диспетчеризации заканчивается, а его возвращаемое значение используется как возвращаемое значение обработчика «верхнего уровня», определяемого формой `defroutes`.

Маршруты в Compojure определяют и обработчик Ring, и шаблон, который используется для выяснения соответствия запроса данному обработчику. Каждый маршрут включает в себя, перечисленное ниже.

- HTTP-метод запросов, обрабатываемых обработчиком. В данном случае методы являются макросами из библиотеки Compojure, имена которых соответствуют названиям HTTP-методов, такие как `GET` и `POST`[2].
- Шаблон URI, описывающий значение слота `:uri`, для которого должен быть вызван обработчик.

---

[1] Так как велика вероятность прийти к неверным выводам, с нашей стороны было бы небрежностью не предложить вам прочитать статью Тима Бернерса-Ли (Tim Berners-Lee) «Cool URIs don't change» («Популярные URI не меняются»): http://www.w3.org/Provider/Style/URI.html.

[2] В число макросов, обозначающих методы запросов, предоставляемых библиотекой Compojure, входят также `PUT`, `DELETE`, `HEAD` и специальный макрос `ANY`, который действует как шаблонный символ (wildcard). При необходимости допускается определять свои макросы для «нестандартных» HTTP-методов, таких как `COPY`.

- Форма привязки запроса и/или параметров, и части URI, совпавшей с шаблоном; имена в этой форме станут локальными привязками в теле обработчика.
- Тело обработчика, содержащее произвольный код на языке Clojure, которое должно вернуть `nil` (чтобы показать, что запрос не был обработан и его маршрутизация должна быть продолжена) или некоторое другое значение, которое можно использовать как основу для ассоциативного массива ответа.

Рассмотрим один из маршрутов:

```
(PUT "/:id" [id url] (retain url id))
```

Этому маршруту соответствуют только запросы со значением `:put` в слоте `:request-method` (что соответствует методу `PUT` протокола HTTP) и с единственным сегментом в URI; например, этому маршруту соответствует URI с сегментом `"/some-id"`, а URI `"/path/to/some-id"` – нет. Элемент `:id` в строке шаблона URI обеспечивает привязку части URI к локальному символу `id`, создаваемому вектором привязки `[id url]`. Локальная привязка `url` не упоминается в строке шаблона; ей присваивается параметр `:url` (или `nil`, если такой параметр в запросе отсутствует). Поведение обработчика и и возвращаемое значение определяются кодом на языке Clojure, следующим за формой привязки.

Так как маршруты Compojure преобразуются в функции-обработчики Ring, мы легко можем проверить в оболочке REPL, как они работают, и поэкспериментировать со всеми их возможностями[1]. Например, Ниже приводится тот же самый маршрут, что используется в реализации службы сокращения URL (хотя здесь он имеет другое тело). Мы можем ясно увидеть, что параметры и именованные сегменты пути в URI привязываются к символам с теми же именами в векторе привязки:

```
((PUT "/:id"
      [id url]
```

[1] Запросы Ring в целом имеют простую конструкцию, и мы создаем их здесь вручную. Если вам постоянно приходится «подделывать» запросы Ring (для тестирования, экспериментов или других целей), вам наверняка пригодится библиотека ring-mock: https://github.com/weavejester/ring-mock.

```
      (list "You requested that " url " be assigned id " id))
  {:uri "/some-id" :params {:url "http://clojurebook.com"} :request-method
:put})
;= {:status 200, :headers {"Content-Type" "text/html"},
;=  :body ("You requested that " "http://clojurebook.com" " be assigned id "
;=  "some-id")}
```

> **Примечание.** Обратите внимание, что обработчики, создаваемые маршрутами Compojure, предусматривают типичные значения по умолчанию для слотов `:status` и `:headers` в ассоциативном массиве ответа. Это позволяет просто возвращать тело ответа, если знаете, что безошибочно производите HTML-страницу.

Сегментам пути, определяемым ключами вида `:keyword`, будут соответствовать любые символы, кроме «/», «.», «,», «;» и «?». Звездочке в шаблоне сегмента пути будут соответствовать все символы до ближайшего символа слеша (/).

Вы можете указать в шаблоне URI любое количество сегментов, и связать любые из них с локальными символами в теле обработчика[1]. Если указать в шаблоне несколько сегментов с одинаковыми именами, соответствующим локальным привязкам будут присвоены векторы с сегментами:

```
((PUT ["/*/*/:id/:id"]
      [* id]
      (str * id))
  {:uri "/abc/xyz/foo/bar" :request-method :put})
;= {:status 200, :headers {"Content-Type" "text/html"},
;= :body "[\"abc\" \"xyz\"][\"foo\" \"bar\"]"}
```

Одной из самых полезных особенностей является поддержка регулярных выражений в определениях сегментов пути. Она близко напоминает особенность `:constraints` в Rails, основанную на регулярных выражениях. Чтобы воспользоваться ею, необходимо заключить в вектор шаблон URI и добавить пары ключ/значение, определяющие имена сегментов и соответствующие им регулярные выражения. Например, с помощью регулярного выражения `#"\d+"` можно указать, что сегмент `:id` может содержать только цифры:

---

[1] Кроме того, библиотека Compojure добавляет в запрос ассоциативный массив `:route-params`, содержащий сегменты пути.

```
((PUT ["/:id" :id #"\d+"]
      [id url]
      (list "You requested that " url " be assigned id " id))
  {:uri "/some-id"                                                    ❶
   :params {:url "http://clojurebook.com"}
   :request-method :put})
;= nil
((PUT ["/:id" :id #"\d+"]
      [id url]
      (list "You requested that " url " be assigned id " id))
  {:uri "/590"                                                        ❷
   :params {:url "http://clojurebook.com"}
   :request-method :put})
;= {:status 200, :headers {"Content-Type" "text/html"},
;= :body "You requested that http://clojurebook.com be assigned id 590"}
```

❶ Запрос не соответствует определению маршрута, потому что сегмент `some-id` пути `:uri` не является числом. Поэтому обработчик возвращает `nil` (сигнализируя, что маршрутизация запроса должна быть продолжена).

❷ Первый сегмент пути `:uri` состоит только из цифр, он соответствует шаблону и поэтому вызывается тело обработчика, возвращающее непустой ответ.

Наконец, вместо формы привязки в маршруте всегда можно использовать символ или ассоциативный массив деструктуризации. Это позволит связать весь ассоциативный массив запроса для использования в теле обработчика:

```
((PUT "/:id" req (str "You requested: " (:uri req)))
  {:uri "/foo" :request-method :put})
;= {:status 200, :headers {"Content-Type" "text/html"},
;=  :body "You requested: /foo"}
((PUT "/:id" {:keys [uri]} (str "You requested: " uri))
  {:uri "/foo" :request-method :put})
;= {:status 200, :headers {"Content-Type" "text/html"},
;=  :body "You requested: /foo"}
```

Последний маршрут в примере 16.2 – это универсальный обработчик, позволяющий определить, что должно произойти, если запрос не совпал ни с одним из маршрутов в форме `defroutes`. Здесь мы возвращаем HTTP-код состояния 404, с телом ответа, определяемым

обработчиком с помощью вызова вспомогательной функции compojure.route/not-found; в данном случае телом является простая строка.

Приложение закончено, однако оно пока не работает! Исправить это несложно. Во-первых, поскольку мы зависим от параметров запроса, которые должны добавляться в запросы, нам необходима некоторая промежуточная функция, анализирующая параметры. Библиотека Compojure содержит две вспомогательные функции в своем пространстве имен compojure.handler, выполняющие эту работу: функция api позволяет обернуть обработчик промежуточной функцией, которая позаботится обо всех параметрах, и она лучше всего подходит для использования в HTTP-службах; функция site обеспечивает те же возможности, но добавляет еще одну промежуточную функцию поддержки дополнительных возможностей, которые обычно ожидаются от веб-сайта, такие как обработка cookies, сеансов и форм, состоящих из нескольких частей, выгрузка файлов и так далее. Воспользуемся первой из них:

```
(require 'compojure.handler)

(def app (compojure.handler/api app*))
```

Теперь осталось лишь запустить приложение с помощью адаптера Jetty:

```
(use '[ring.adapter.jetty :only (run-jetty)])
;= nil
(def server (run-jetty #'app {:port 8080 :join? false}))
;= #'user/server
```

Приложение будет доступно из любого веб-браузера, но оно построено настолько правильно, что его можно использовать программно, с помощью любой библиотеки поддержки HTTP. Чтобы в полном объеме протестировать получившуюся службу сокращения адресов URL, нужно попробовать отправить ей запросы PUT или POST с параметрами, настроенными должным образом. Сделать это с помощью браузера будет сложно, так как ему нужна форма, чтобы отправить запрос любого из этих типов. Однако существует широко распространенный инструмент командной строки curl, с помощью которого можно относительно просто протестировать нашу службу. curl позволяет указать HTTP-метод с помощью ключа -X, и вывести полный ответ, полученный от сервера, если указать ключ -i.

Попробуем создать пару коротких идентификаторов URL, послав запрос PUT:

```
% curl -X PUT 'http://localhost:8080/sicp?url=http://mitpress.mit.edu/sicp/'
URL http://mitpress.mit.edu/sicp/ assigned the short identifier sicp

% curl -X PUT 'http://localhost:8080/clj?url=http://clojure.org'
URL http://clojure.org assigned the short identifier clj
```

Если добавить ключ `-i`, `curl` выведет заголовки, полученные от нашей службы. Если сделать это, посылая запрос POST, можно будет убедиться, что служба корректно вернула HTTP-код состояния `201 Created` и соответствующий заголовок `Location`:

```
% curl -i -X POST 'http://localhost:8080/?url=http://clojurebook.com'
HTTP/1.1 201 Created
Date: Sun, 18 Dec 2011 20:58:09 GMT
Location: 1
Content-Length: 58
Server: Jetty(6.1.25)
URL http://clojurebook.com assigned the short identifier 1
```

Если теперь попытаться зарегистрировать короткий идентификатор, который уже занят, служба ответит HTTP-кодом ошибки `409 Conflict`, который может использоваться программным агентом, чтобы вывести пояснительное сообщение:

```
% curl -i -X PUT 'http://localhost:8080/1?url=http://apple.com'
HTTP/1.1 409 Conflict
Date: Sun, 18 Dec 2011 20:58:40 GMT
Content-Length: 28
Server: Jetty(6.1.25)
Short URL 1 is already taken
```

Зарегистрировав несколько адресов URL, можно попробовать использовать маршрут `/list/`, чтобы получить список известных коротких идентификаторов:

```
% curl http://localhost:8080/list/
1
clj
sicp
```

В ответ на обращение к любому URI, не соответствующему ни одному из маршрутов (или с одним сегментом пути, не совпадающим ни с одним из известных коротких идентификаторов URL) корректно возвращается ответ `404 Not Found` и сообщение:

```
% curl -i http://localhost:8080/foo
HTTP/1.1 404 Not Found
Date: Sun, 18 Dec 2011 21:21:39 GMT
Content-Length: 22
Server: Jetty(6.1.25)
No such short URL: foo
```

А в ответ на обращение к любому не короткому идентификатору URL возвращается обобщенный ответ `404`:

```
% curl -i http://localhost:8080/some/other/url
HTTP/1.1 404 Not Found
Date: Sun, 18 Dec 2011 21:21:53 GMT
Content-Length: 28
Server: Jetty(6.1.25)
Sorry, there's nothing here.
```

Наконец, можно проверить, как выполняется переадресация в ответ на запрос, содержащий известный короткий идентификатор URL:

```
% curl -i http://localhost:8080/sicp
HTTP/1.1 302 Found
Date: Sun, 18 Dec 2011 20:59:12 GMT
Location: http://mitpress.mit.edu/sicp/
Content-Length: 0
Server: Jetty(6.1.25)

% curl -L http://localhost:8080/sicp    ❶
<HTML><HEAD><TITLE>Welcome to the SICP Web Site</TITLE></HEAD>
....
```

❶ Ключ `-L` сообщает команде `curl`, что при получении HTTP-кода `302 Found` она должна выполнить переход по адресу, указанному в заголовке `Location`.

Ниже приводится полная реализация нашей службы сокращения адресов URL, как она должна выглядеть в исходном файле.

Пример 16.3. Служба сокращения адресов URL

```
(ns com.clojurebook.url-shortener
  (:use [compojure.core :only (GET PUT POST defroutes)])
  (:require (compojure handler route)
            [ring.util.response :as response]))

(def ^:private counter (atom 0))

(def ^:private mappings (ref {}))

(defn url-for
  [id]
  (@mappings id))

(defn shorten!
  "Сохраняет указанный URL под новым или под указанным идентификатором.
   Возвращает идентификатор в виде строки. Изменяет соответственно
   глобальный ассоциативный массив."
  ([url]
   (let [id (swap! counter inc)
         id (Long/toString id 36)]
     (or (shorten! url id)
         (recur url))))
  ([url id]
    (dosync
      (when-not (@mappings id)
        (alter mappings assoc id url)
        id))))

(defn retain
  [& [url id :as args]]
  (if-let [id (apply shorten! args)]
    {:status 201
     :headers {"Location" id}
     :body (list "URL " url " assigned the short identifier " id)}
    {:status 409 :body (format "Short URL %s is already taken" id)}))

(defn redirect
  [id]
  (if-let [url (url-for id)]
    (response/redirect url)
    {:status 404 :body (str "No such short URL: " id)}))

(defroutes app*
```

```
  (GET "/" request "Welcome!")
  (PUT "/:id" [id url] (retain url id))
  (POST "/" [url] (if (empty? url)
                    {:status 400 :body "No `url` parameter provided"}
                    (retain url)))
  (GET "/:id" [id] (redirect id))
  (GET "/list/" [] (interpose "\n" (keys @mappings)))
  (compojure.route/not-found "Sorry, there's nothing here."))

(def app (compojure.handler/api app*))

;; ; Чтобы запустить на локальном компьютере:
;; (use '[ring.adapter.jetty :only (run-jetty)])
;; (def server (run-jetty #'app {:port 8080 :join? false}))
```

Комбинирование маршрутов. Так как форма `defroutes` может вызывать любые функции-обработчики Ring и сама создает функцию-обработчик, вы можете составлять целые иерархии маршрутов практически без всяких усилий. Если бы нам потребовалось добавить в нашу службу административную страницу, мы легко могли бы добавить соответствующий маршрут в существующий набор маршрутов `app*`:

```
(defroutes app+admin
  (GET "/admin/" request ...)
  (POST "/admin/some-admin-action" request ...)
  app*)
```

Единственное, о чем следует помнить: не забывайте удалять «универсальные» маршруты (например, как тот, что использует функцию `compojure.route/not-found` в нашем примере) из групп маршрутов, которые предполагается добавлять в начало группы верхнего уровня; в противном случае маршруты, следующие ниже, не будут участвовать в процессе маршрутизации.

Обработка шаблонов

К настоящему моменту мы реализовали почти все, что можно было бы ожидать от веб-фреймворка, с одним важным исключением – отсутствует возможность воспроизводить сложные HTML-страницы. Хотя HTTP-службы и файловые серверы полезны сами по себе, но большинство людей сразу представляют себе «HTML»,

когда слышат словосочетание «веб-приложение». В большинстве веб-фреймворков используются системы обработки шаблонов, позволяющие смешивать разметку HTML с выполняемым кодом или «директивами», ссылающимися на переменные, хранящие информацию для страницы, и воспроизводящие заполненные HTML-страницы.

Одним из примеров таких систем может служить язык шаблонов ERB в Ruby:

Пример 16.4. Пример HTML-шаблона на языке ERB

```
<h1>Hello, <%= @user.name %></h1>

<p>These are your friends:</p>
<ul>
<% @user.friends.each do |friend| %>
  <li><%= friend.name %></li>
<% end %>
</ul>
```

Незнакомые с синтаксисом ERB могут прийти в замешательство, но в действительности он не сильно отличается от синтаксиса шаблонов Django, или JSP, или любой из сотен других систем обработки шаблонов, которые могут быть вам знакомы. Все эти системы основаны, прежде всего, на операции подстановки строк. В шаблонах ERB выполняемый код на языке Ruby помещается в разделители `<%` и `%>`, а выражения, возвращающие строки для включения в документ – в разделители `<%=` и `%>`. Шаблоны ERB обрабатываются в определенном контексте, поэтому такие выражения, как `@user.name`, ссылаются на локальные переменные контекста и возвращают значения указанных атрибутов.

Такой подход уже многие годы используется в самых разных языках миллионами программистов; более того, это основной режим работы в PHP. Однако не все так гладко. Первая очевидная проблема в том, что разметка HTML в этих шаблонах сложна в отладке и тонкой настройке; в отсутствие рабочего веб-стека невозможно убедиться в работоспособности шаблонов HTML или внести изменения в таблицы стилей. Похожая проблема наблюдается и в программном коде, потому что его сложно протестировать отдельно, без использования нетривиального фиктивного окружения с соответствующими объектами и прочими элементами. Выражаясь точнее, модель и представ-

ление образуют более тесную связь, чем нам хотелось бы. Наконец, разработчик таких шаблонов в идеале должен быть экспертом и в HTML, и в Ruby (или Python, или PHP), и хорошо разбираться в программном стеке, управляющим шаблонами.

Несмотря на популярность решений на основе шаблонов, крайне редко встречаются специалисты, которые одновременно являются экспертами и в веб-дизайне, и в разработке программного обеспечения. Обычно дизайнер занимается созданием заготовок для HTML-страниц, а разработчик редактирует их вручную, добавляя разметку используемого языка шаблонов. Если в дизайн вносятся какие-то изменения, очень часто это приводит к изменению первоначальных HTML-документов, которые затем разработчик должен тщательно проверить и объединить с шаблонизированными версиями. Совершенно очевидно, что это далеко не оптимальный сценарий. Существует ли более удачное решение?

Enlive: преобразование HTML с применением селекторов

Библиотека Enlive (http://github.com/cgrand/enlive) предлагает радикальный способ отделения кода от шаблонов: вместо определения специального локального синтаксиса для интерполяции значений в шаблоны, Enlive позволяет использовать в качестве шаблонов простые HTML-файлы без специальных тегов, без специальных атрибутов, без специальных классов и без специального синтаксиса. Динамическое содержимое внедряется в шаблоны кодом на языке Clojure, использующим *селекторы* (во многом напоминающие селекторы CSS) для выбора элементов, которые следует модифицировать, и функции Clojure, реализующие применяемые трансформации[1].

Такое строгое разделение, когда представление полностью изолировано от кода, реализующего его преобразование, существенно упрощает совместную работу программистов и дизайнеров Селекторы, используемые для идентификации модифицируемых элемен-

---

[1] Обработка шаблона – это всего лишь частный случай преобразования данных в формате HTML или XML: библиотека Enlive может также использоваться для извлечения содержимого из документов HTML и XML с использованием тех же самых селекторов, что используются при обработке шаблонов. Однако здесь мы обсуждаем применение Enlive только для обработки шаблонов.

тов шаблона, практически такие же, как некоторые селекторы CSS, применяемые для оформления содержимого. Поэтому изменения в шаблоне, способные оказать влияние на код, выполняющий трансформацию шаблона, легко могут быть заблаговременно идентифицированы дизайнером (или программистами, привлеченными к разработке дизайна!).

Попробуем воду

При использовании Enlive обычно сначала создается HTML-файл – который обычно называют *исходным шаблоном* – однако мы предлагаем сначала получить некоторое представление о том, как работает Enlive, использовав для экспериментов оболочку REPL. Добавьте зависимость в свой проект:

```
[enlive/enlive "1.0.0"]
```

...и запустите REPL.

```
(require '[net.cgrand.enlive-html :as h])
;= nil
(h/sniptest "<h1>Lorem Ipsum</h1>")
;= "<h1>Lorem Ipsum</h1>"
```

`sniptest` – это утилита из библиотеки Enlive, упрощающая возможность экспериментов с фрагментами разметки HTML и их преобразования в REPL. Здесь мы не определили никаких преобразований, поэтому фрагмент был возвращен в исходном виде.

```
(h/sniptest "<h1>Lorem Ipsum</h1>"
  [:h1] (h/content "Hello Reader!"))
;= "<h1>Hello Reader!</h1>"
```

`[:h1]` – это селектор, соответствующий CSS-селектору `h1`, которому соответствуют все элементы `h1` в разметке HTML. `content` – это функция высшего порядка, замещающая тело элемента, совпавшего с селектором, значением аргумента. Библиотека Enlive содержит огромное количество готовых функций высшего порядка, удовлетворяющих все основные нужды, возникающие при обработке шаблонов в типичных веб-приложениях, а ее селекторы являются надмножеством селекторов CSS и позволяют выбирать элементы разметки для преобразования.

Задержимся немного и посмотрим, насколько сложные преобразования можно выполнять с помощью библиотеки. В Enlive имеется функция `html-snippet`, позволяющая выполнять парсинг любого содержимого в формате HTML и возвращающая последовательность ассоциативных массивов, представляющих отдельные элементы с атрибутами и содержимым:

```
(h/html-snippet
  "<p>x, <a id=\"home\" href=\"/\">y</a>, <a href=\"..\">z</a></p>")
;= ({:tag :p,
;=   :attrs nil,
;=   :content
;=   ("x, "
;=     {:tag :a, :attrs {:href "/", :id "home"}, :content ("y")}
;=     ", "
;=     {:tag :a, :attrs {:href ".."}, :content ("z")})})
```

Это представление разметки HTML/XML соответствует тому, что производят и принимают функции из пространства имен `clojure.xml` — еще один пример, когда применение обобщенных абстракций позволяет упростить обмен данными и повторно использовать функциональные возможности.

Зная это, нетрудно представить, как реализованы преобразования в Enlive:

1. Селекторы выполняют обход дерева, представляющего документ HTML, пытаясь найти соответствующие им элементы.
2. Найденные элементы разметки передаются для преобразования функциям, связанным с селекторами. Результаты, возвращаемые функциями, замещают эти элементы.

Поскольку операции выполняются над неизменяемыми структурами данных с помощью чистых функций, их можно объединять в цепочки, составлять из них композиции и использоваться повторно, в зависимости от ваших потребностей:

Пример 16.5. Немного более сложное преобразование HTML

```
(h/sniptest
  "<p>x, <a id=\"home\" href=\"/\">y</a>, <a href=\"..\">z</a></p>"
  [:a#home] (h/set-attr :href "http://clojurebook.com")
  [[:a (h/attr= :href "..")]] (h/content "go up"))
;= "<p>x, <a href=\"http://clojurebook.com\" id=\"home\">y</a>, <a
;= href=\"..\">go up</a></p>"
```

Для максимального использования возможностей библиотеки Enlive необходимо понимать, как определять селекторы и функции преобразований.

Селекторы

Первое время синтаксис селекторов в Enlive может казаться пугающим, но в действительности он прост в освоении, особенно если вы знакомы с селекторами CSS. В большинстве случаев можно ограничиться добавлением двоеточий перед каждым элементом селектора и заключением всей последовательности элементов в вектор. Например, CSS-селектор `div span.phone` превращается в `[:div :span.phone]`, `#summary .kw` превращается в `[:#summary :.kw]`, и так далее.

Начинающие осваивать библиотеку Enlive часто испытывают затруднения с вложенными векторами в селекторах. Здесь действует простое правило: *самый внешний вектор обозначает иерархическую цепочку селекторов, все остальные векторы выполняют операцию конъюнкции для своих элементов*[1]. То есть, селектор `[:div [:span :.phone]]` эквивалентен селектору `[:div :span.phone]`, упоминавшемуся выше. Векторы, соответствующие элементам цепочки, могут вкладываться *как угодно*: селектор `[:div [:span [:.phone :.mobile]]]` суть то же самое, что и `[:div :span.phone.mobile]`.

Обратите внимание, что самый внешний вектор не является чем-то необязательным, даже если селектор включает единственный шаг, форма записи `:h1` не будет считаться допустимым селектором, только `[:h1]`.

Библиотека Enlive поддерживает также дизъюнкцию[2]. CSS-селектору `div#info span.phone, div#info span.email` соответствует Enlive-селектор `#{[:div#info :span.phone] [:div#info :span.email]}`. Однако, в отличие от CSS, операцию дизъюнкции можно применять не только на верхнем уровне: `[:div#info #{:span.phone :span.email}]` или даже `[:div#info [:span #{:.phone :.email}]]` – все это разные формы записи одного и того же селектора.

> **Примечание.** Подведем итоги: множества обозначают дизъюнкцию, внутренние векторы обозначают конъюнкцию, внешние векторы обозначают объединение в иерархическую цепочку.

---

[1] Логическая операция «И». – *Прим. перев.*

[2] Логическая операция «ИЛИ». – *Прим. перев.*

Все остальные проверки выполняются с помощью предикатов и также могут быть расширены при желании. Например, предикат `attr?` позволяет выразить CSS-селектор `a[class]`, как `[[:a (attr? :class)]]`. Обратите внимание на вложенность векторов: селектор с единственным вектором – то есть, `[:a (attr? :class)]` – эквивалентен CSS-селектору `a *[class]`. Взгляните на различия:

```
(h/sniptest "<p class=\"\"><a href=\"\" class=\"\"></a></p>"
  [[:p (h/attr? :class)]] (h/content "XXX"))
;= "<p class=\"\">XXX</p>"

(h/sniptest "<p class=\"\"><a href=\"\" class=\"\"></a></p>"
  [:p (h/attr? :class)] (h/content "XXX"))
;= "<p class=\"\"><a class=\"\" href=\"\">XXX</a></p>"
```

Таким образом, библиотека Enlive поддерживает большинство CSS-селекторов (включая все псевдоклассы `:nth-*`). Кроме того, имеется возможность определять собственные селекторы в виде обычных функций. Для этого можно использовать функции высшего порядка `pred` и `zip-pred` из библиотеки Enlive, принимающие предикаты для элементов и предикаты для зипперов (zippers)[1], соответственно, и возвращающие функцию, способную играть роль элемента селектора.

Предикат `attr=` в библиотеке Enlive позволяет определять селекторы для элементов с определенным значением указанного атрибута[2]. Попробуем определить новую функцию элемента селектора для элементов, имеющих любые атрибуты с указанным значением:

```
(defn some-attr=
  "Элемент селектора, соответствует любым элементам хотя бы с одним
   атрибутом, имеющим указанное значение."
  [value]
  (h/pred (fn [node]
            (some #{value} (vals (:attrs node))))))
```

Посмотрим, как она работает:

```
(h/sniptest "<ul><li id=\"foo\">A<li>B<li name=\"foo\">C</li></ul>"
  [(some-attr= "foo")] (h/set-attr :found "yes"))
```

[1] Зипперы обсуждались в разделе «Навигация, изменение и зипперы (zippers)», в главе 3.

[2] Предикат `attr=` использовался в примере 16.5.

```
;= "<ul>
;=     <li found=\"yes\" id=\"foo\">A</li>
;=     <li>B</li>
;=     <li found=\"yes\" name=\"foo\">C</li>
;= </ul>"
```

Библиотека Enlive уже включает массу гибких инструментов управления преобразованиями. Как было показано выше, когда имеющейся гибкости оказывается недостаточно, легко можно определять собственные селекторы, используя любые критерии по своему выбору.

Итерации и ветвление

Теперь мы знаем, как идентифицировать узлы и как преобразовывать их (например, с помощью `content` или `set-attr`), но мы пока не коснулись двух важных аспектов обработки шаблонов: условных инструкций и итераций.

Ключом к организации итераций и ветвления в Enlive является понимание, что преобразования могут быть:

- функциями, принимающими и возвращающими один элемент;
- функциями, принимающими один элемент и возвращающими коллекцию элементов;
- значением `nil`, который является эквивалентом `(fn [_] nil)`.

Из этого следует, что, например, вывод необязательного сообщения можно организовать с помощью формы `when`, возвращающей `nil`, если условное выражение имеет ложное значение:

```
(defn display
  [msg]
  (h/sniptest "<div><span class=\"msg\"></span></div>"
    [:.msg] (when msg (h/content msg))))
;= #'user/display
(display "Welcome back!")
;= "<div><span class=\"msg\">Welcome back!</span></div>"
(display nil)
;= "<div></div>"
```

Когда сообщение присутствует, форма `when` вызывает функцию преобразования `(h/content msg)`, которая запишет указанное сообщение в соответствующий элемент `span`. С другой стороны, если сообщение отсутствует, форма `when` вернет `nil` и элемент, служащий контейнером для сообщения, будет удален.

В некоторых случаях может оказаться желательным оставить пустой элемент span, потому что он может быть необходим клиентской части веб-приложения; в этом случае вместо формы when можно использовать if (или cond, или любую другую условную форму):

```
(defn display
  [msg]
  (h/sniptest "<div><span class=\"msg\"></span></div>"
    [:.msg] (if msg
              (h/content msg)
              (h/add-class "hidden"))))
;= #'user/display
(display nil)
;= "<div><span class=\"msg hidden\"></span></div>"
```

Итерации в Enlive выполняются с помощью формы clone-for, которая выглядит и действует точно так же, как форма for:

```
(defn countdown
  [n]
  (h/sniptest "<ul><li></li></ul>"
    [:li] (h/clone-for [i (range n 0 -1)]
            (h/content (str i)))))
;= #'user/countdown
(countdown 0)
;= "<ul></ul>"
(countdown 3)
;= "<ul><li>3</li><li>2</li><li>1</li></ul>"
```

За кулисами форма for генератора списков возвращает последовательность функций преобразования (в примере выше – n экземпляров (h/content (str i))), каждая из которых создает элемент(ы) на основе единственного узла, выбранного селектором. Полученные элементы замещают оригинальный узел.

Когда дело доходит до итераций, часто бывает необходимо удалить некоторые атрибуты, использовавшиеся как критерий для выбора узла. Например, атрибут id. Сделать это можно с помощью функции do->, которая принимает преобразования и применяет их по очереди:

```
(defn countdown
  [n]
```

```
  (h/sniptest "<ul><li id=\"foo\"></li></ul>"
    [:#foo] (h/do->
              (h/remove-attr :id)
              (h/clone-for [i (range n 0 -1)]
                (h/content (str i))))))
;= #'user/countdown
(countdown 3)
;= "<ul><li>3</li><li>2</li><li>1</li></ul>"
```

Функцию `do->` можно использовать везде, где ожидается функция преобразования, потому что она сама возвращает функцию преобразования. Благодаря этому вы можете составлять любые композиции из функций преобразований.

Объединяем все вместе

Функция `sniptest` является отличным подспорьем в разработке, так как позволяет исследовать все основы библиотеки Enlive, но в действующих приложениях она не имеет большой практической ценности. В приложении нам нужно загрузить HTML-файл с диска, точнее, из каталога, находящегося в списке путей поиска classpath приложения. Эта работа возлагается на формы `deftemplate` и `defsnippet`.

Форма `defsnippet` определяет функцию, загружающую HTML-файл, который находится где-то в пути поиска classpath, и преобразует его подобно функции `sniptest`. Эти функции предназначены для вызова из других функций, созданных с помощью формы `defsnippet` или `deftemplate`, что обеспечивает простой способ компоновки содержимого из составных элементов. Например, допустим, что у нас имеется файл *footer.html* в пути поиска classpath:

Пример 16.6. footer.html

```
<div class="footer"/>
```

мы можем объявить фрагмент `footer` многократного пользования:

```
(h/defsnippet footer "footer.html" [:.footer]      ❶
  [message]                                        ❷
  [:.footer] (h/content message))                  ❸
```

❶ `footer` – это имя объявляемой переменной и функции. `"footer.html"` – это путь к HTML-файлу, откуда загружается содержимое, и может

быть строкой или экземпляром java.io.File, java.net.URL или java.net.URI. Третий аргумент формы defsnippet – это селектор, определяющий корневой элемент в загруженном HTML-файле, к которому будут применяться преобразования. Селектор [:.footer], используемый здесь, гарантирует отсутствие элементов <html> и <body>, которые библиотека Enlive добавляет неявно при загрузке фрагментов и шаблонов. Один HTML-файл может содержать несколько фрагментов, каждая форма defsnippet выберет только один соответствующий узел – довольно удобно иметь возможность хранить несколько компонентов многократного пользования в одном файле и просматривать их, используя только веб-браузер.

❷ Вектор аргументов; эта функция преобразования фрагмента принимает единственный аргумент, message.

❸ Остальная часть формы defsnippet содержит пары селекторов и функций преобразования, как уже было показано выше.

Примечание. При использовании инструмента сборки проектов Leiningen, шаблоны HTML лучше сохранять в каталоге resources (значение по умолчанию параметра :resources-path в project.clj). При использовании Maven, шаблоны HTML обычно принято сохранять в дереве каталогов с корнем в каталоге src/main/resources.

В отличие от sniptest, функция, созданная с помощью defsnippet, возвращает последовательность ассоциативных массивов, представляющих элементы HTML:

```
(footer "hello")
;= ({:tag :div, :attrs {:class "footer"}, :content ("hello")})
```

Форма deftemplate работает практически так же, но она не позволяет определить корневой элемент для применения преобразований, и вместо последовательности ассоциативных массивов с элементами HTML, функции, объявленные с помощью deftemplate, возвращают «ленивую» последовательность строк с фрагментами HTML, которые могут использоваться в элементе :body ассоциативного массива ответа для библиотеки Ring.

Теперь мы легко сможем повторить шаблон ERB, представленный в примере 16.4. Но сначала определим файл шаблона:

Пример 16.7. friends.html

```
<h1>Hello, <span class="username"/></h1>
<p>These are your friends:</p>
<ul class="friends"><li/></ul>
```

Далее, либо в приложении, либо в оболочке REPL можно определить функцию шаблона Enlive и воспроизвести с ее помощью законченный документ HTML:

```
(h/deftemplate friends-list "friends.html"
  [username friends]
  [:.username] (h/content username)
  [:ul.friends :li] (h/clone-for [f friends]
                      (h/content f)))

(friends-list "Chas" ["Christophe" "Brian"])                    ❶
;= ("<html>" "<body>" "<h1>" "Hello, " "<span class=\"username\">"
;=  "Chas" "</span>" "</h1>" "\n" "<p>These are your friends:</p>"
;=  "\n" "<ul class=\"friends\">" "<li>" "Christophe" "</li>" "<li>"
;=  "Brian" "</li>" "</ul>" "\n" "</body>" "</html>")
```

❶ Не забывайте, что функция, объявленная с помощью `deftemplate`, возвращает последовательность строк. Поэтому, чтобы вернуть содержимое клиенту с помощью библиотеки Ring, необходимо объединить эти строки в одну строку, например, вызовом `(apply str (friendslist ...))`.

Итак, мы повторили шаблон ERB, рассматривавшийся выше, но, похоже, что нам не удалось уйти далеко. Шаблон ERB, несмотря на имеющиеся проблемы, имеет явное преимущество – он более краток. Даже при том, что код ERB встроен в разметку HTML, все равно создается ощущение, что реализация шаблона с применением библиотеки Enlive требует больше кода. Это можно считать вполне приемлемой ценой, учитывая преимущества, полученные в других аспектах. В конце концов, ни один подход нельзя назвать совершенным.

Настоящие выгоды от использования Enlive в полной мере начинают ощущаться, когда требования к шаблонам и функциональным возможностям становятся далеко не тривиальными. Поскольку все операции в Enlive реализуются как функции, манипулирующие стандартными структурами данных, мы легко можем объединять их. В этом заключается резкое отличие от большинства систем обработки шаблонов, работающие на уровне строк и операции конкатенации строк.

Для демонстрации добавим новый класс к каждому элементу списка в нашем примере, воспользовавшись формой `do->` для объединения двух функций преобразования:

```
(h/deftemplate friends-list "friends.html"
  [username friends friend-class]
  [:.username] (h/content username)
  [:ul.friends :li] (h/clone-for [f friends]
                      (h/do-> (h/content f)
                              (h/add-class friend-class))))

(friends-list "Chas" ["Christophe" "Brian"] "programmer")
;= ("<html>" "<body>" "<h1>" "Hello, " "<span class=\"username\">" "Chas"
;=  "</span>" "</h1>" "\n" "<p>These are your friends:</p>" "\n"
;=  "<ul class=\"friends\">" "<" "li" " " "class" "=\"" "programmer" "\""
;=  ">" "Christophe" "</" "li" ">" "<" "li" " " "class" "=\"" "programmer"
;=  "\"" ">" "Brian" "</" "li" ">" "</ul>" "\n" "</body>" "</html>")
```

Мы лишь немного увеличили объем кода, добавив единственное изменение, и это изменение выполнено только в программном коде. То есть HTML-файл шаблона остался нетронутым. Для сравнения, традиционный метод ERB начинает приводить к ухудшению удобочитаемости:

```
<h1>Hello, <%= @user.name %></h1>

<p>These are your friends:</p>
<ul>
<% @user.friends.each do |friend| %>
  <li class="<%= @friendclass %>"><%= friend.name %></li>
<% end %>
</ul>
```

Наконец, добавим «подвал» (footer) в страницу:

```
(h/deftemplate friends-list "friends.html"
  [username friends friend-class]
  [:.username] (h/content username)
  [:ul.friends :li] (h/clone-for [f friends]
                      (h/do-> (h/content f)
                              (h/add-class friend-class)))
  [:body] (h/append (footer (str "Goodbye, " username))))        ❶

(friends-list "Chas" ["Christophe" "Brian"] "programmer")
;= ("<html>" "<body>" "<h1>" "Hello, " "<span class=\"username\">" "Chas"
;=  "</span>" "</h1>" "\n" "<p>These are your friends:</p>" "\n"
;=  "<ul class=\"friends\">" "<" "li" " " "class" "=\"" "programmer" "\""
```

```
;=  ">" "Christophe" "</" "li" ">" "<" "li" " " "class" "=\"" "programmer"
;=  "\"" ">" "Brian" "</" "li" ">" "</ul>" "\n" "<div class=\"footer\">"
;=  "Goodbye, Chas" "</div>" "</body>" "</html>")
```

❶ В отличие от функции `content`, замещающей содержимое выбранного элемента, функция `append` добавляет новое содержимое в конец. Если бы здесь использовалась `content`, разметка «подвала» оказалась бы единственным содержимым элемента `<body>` в сгенерированном HTML-документе.

Функции, объявленные с помощью `defsnippet`, можно вызывать непосредственно из функций обработки шаблонов, как было показано выше; их и их результаты можно передавать функциям обработки шаблонов в виде аргументов; или даже находить и вызывать функции создания фрагментов, опираясь на имена классов в разметке HTML, в файлах шаблонов. Поскольку все сущности, вовлеченные в создание шаблонов Enlive, являются универсальными (функциями, принимающими и возвращающими коллекции, поддерживающие обобщенные абстракции), вы свободно можете выбирать способ объединения шаблонов и фрагментов для сборки полных страниц.

В заключение

Повсюду в этой главе мы видели, как применение функциональных подходов к проектированию программного обеспечения порождает небольшие, но очень мощные абстракции, позволяющие быстро создавать приложения, выполняющиеся на стороне сервера. Для реализации HTTP-службы нам потребовалось написать всего 26 коротких строк. Мы показали радикальной иной подход к обработке веб-содержимого с помощью библиотеки Enlive, отделив шаблоны от программного кода. Мы познакомились со всеми компонентами, которые могут пригодиться для создания небольшого по величине веб-фреймворка. При наличии подходящего хранилища, остается не так много особенностей, которые нельзя было бы реализовать с помощью этих компонентов.

Но самое интересное не то, что мы показали, а то, чего не показали: большой стек из фреймворков и генераторов. Это можно было бы считать недостатком комплекса веб-библиотек для Clojure. Пришедшие из мира Python или Ruby (и даже Java) могли привыкнуть к использованию генераторов, креплений (fixtures), контроллеров и

представлений. Ничего этого нет в наших примерах, и их отсутствие может оказаться непривычным для опытных веб-разработчиков.

Опытный функциональный программист заметит, что такое положение дел характерно практически для всех функциональных языков программирования. Благодаря природе функционального программирования, сложнейшие системы могут быть сконструированы из очень простых функций. Например, чтобы добавить авторизацию в наш пример, достаточно реализовать несложную промежуточную функцию. Легковесная природа вовлекаемых библиотек и абстракций, делает это возможным и простым, в противоположность весьма тяжеловесным механизмам авторизации в полноценных веб-стеках, таких как Django или Spring.

Легковесность функционального подхода произвела такое впечатление, что он начал оказывать серьезное влияние на сообщества веб-разработчиков, использующих традиционные решения. Практически в каждом языке стали появляться маленькие, «сверхлегковесные» веб-фреймворки, реализующие только простейшую маршрутизацию, отображение и моделирование.

Нельзя сказать, что библиотеки Ring, Compojure и Enlive являются последним словом в веб-разработке на Clojure. Стали появляться и комплексные фреймворки[1], а также различные крупные фреймворки и серверы приложений[2]. Однако большинство этих новейших разработок опираются на элементы, исследованные здесь.

---

[1] Такие как Noir (http://www.webnoir.org) и Ringfinger (https://github.com/myfreeweb/ringfinger).

[2] Наиболее заметным из них является Immutant, сервер приложений для Clojure, построенный на основе JBoss: http://immutant.org.

Глава 17. Развертывание веб-приложений на Clojure

Достигнув некоторого уровня компетентности в Clojure и приближаясь к завершению работы над приложением, вы неизбежно столкнетесь с необходимостью предоставить доступ к его функциональности своим пользователям и клиентам. В современную эпоху распространение приложений все чаще сводится к развертыванию их на стороне сервера (или «в облаке») и организации взаимодействий с ними через веб-службы и веб-интерфейсы. В этой главе мы исследуем различные способы упаковки и последующего развертывания веб-приложений на Clojure, с использованием надежных инструментов, предоставляемых виртуальной машиной JVM и экосистемой Java[1].

Веб-архитектура Java и Clojure

Почти все веб-приложения на Clojure упаковываются и развертываются как *сервлеты* (servlets), та же основная архитектура используется веб-приложениями на Java. Веб-сервлеты – это обычные Java-классы, наследующие базовый класс `javax.servlet.http.HttpServlet`, который определяет программный интерфейс для обработки HTTP-запросов. Этот класс определяет методы для всех типов HTTP-запросов (`GET`, `POST` и так далее), каждый из которых принимает объекты запроса и ответа; каждый метод обработки HTTP-

---

[1] Приемы и инфраструктура, описываемые здесь (с небольшими изменениями), с тем же успехом могут использоваться для развертывания и управления серверными приложениями, не экспортирующими веб-службы. Более того, клиентские приложения на Clojure (для использования на настольных компьютерах или в мобильных устройствах) в общем случае можно развертывать аналогичным способом, что и веб-приложения: посмотрите, как развертываются подобные приложения на Java, и используйте те же приемы и инфраструктуру.

запросов исследует входящий запрос и координирует заполнение объекта ответа. Приложения, следующие спецификации сервлетов (требования которой сводятся к реализации единственного Java-интерфейса и соблюдению некоторых соглашений об упаковке), могут развертываться как веб-приложения на любых из множества серверов приложений, большинство которых могут предложить особые возможности, помимо базовой поддержки сервлетов Java (такие как организация пулов соединений с базами данных, очереди сообщений, средства управления и мониторинга, и так далее).

За редкими исключениями, серверы приложений поддерживают возможность размещения в них множества приложений, каждое из которых может содержать несколько сервлетов (рис. 17.1). Практически все серверы приложений одновременно являются веб-серверами (часто довольно зрелыми, высококачественными реализациями серверов HTTP/HTTPS), но есть также возможность развернуть приложение на сервере приложений, доступ к которому осуществляется посредством выделенного веб-сервера (таком как Apache httpd, lighttpd или IIS)[1].

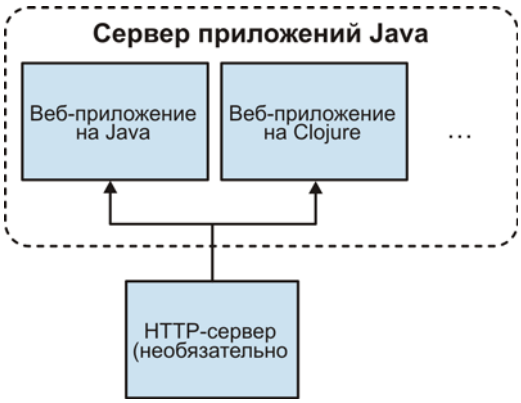

Рис. 17.1. Веб-архитектура Java

[1] Некоторые серверы приложений предлагают другие методы взаимодействий между экземпляром сервера приложений и выделенным веб-сервером, кроме проксирования HTTP; в высоконагруженных системах такие методы могут оказаться более эффективными, чем проксирование HTTP. Например, сервер приложений Tomcat предлагает набор модулей для Apache и ISAPI, реализующие протокол обмена компактными двоичными данными: http://tomcat.apache.org/connectors-doc/.

Опираясь на эту архитектуру и экосистему, веб-приложения на Clojure могут пользоваться тем же многообразием возможностей развертывания и операционной среды, что и веб-приложения на Java.

> **Примечание.** Перечень спецификаций серверов Java достаточно обширен, и сервлеты являются в нем лишь одним из пунктов (одним из основных, по общему признанию). В числе других пунктов можно назвать стандарты поддержки очередей сообщений, доступа к каталогам и базам данных, определяемые спецификациями JMS, JNDI и JPA, соответственно. Если в вашей организации уже используются эти механизмы, вам будет приятно узнать, что Clojure поддерживает и может использовать их, так же как и сервлеты. С другой стороны, если прежде вы не слышали об этих стандартах JEE (Java Enterprise Edition, он же J2EE), вам не менее приятно будет узнать, что из знание не требуется для успешной сборки и развертывания полнофункциональных веб-приложений на Clojure.

Сервлеты можно реализовать непосредственно на языке Clojure (в конце концов, для этого достаточно унаследовать базовый класс `HttpServlet`), но гораздо предпочтительнее использовать для этого веб-фреймворк на Clojure (такой как Ring, подробно рассматривавшийся в главе 16), чтобы абстрагироваться от программной (и в значительной степени императивной) природы API сервлетов. В случае применения Ring, каждый обработчик является функцией, принимающей ассоциативный массив Clojure в аргументе, являющийся идиоматическим представлением данных в объекте HTTP-запроса сервлета, и возвращающей значение, которое адаптер Ring запишет в объект HTTP-ответа сервлета. Затем обработчики объединяются в маршруты – совмещающие анализ HTTP-метода запроса (такого как `GET`) и адреса URL – чтобы в результате получить единственную функцию, инкапсулирующую функциональность приложения для обработки всех типов HTTP-методов и адресов URL. Библиотека Ring поддерживает несколько адаптеров, делегирующих вызовы конкретных методов сервлетов этой функции.

Если «приблизить» блок с веб-приложением на Clojure (рис. 17.1), взаимоотношения между сервлетами и библиотекой Ring можно изобразить, как показано на рис. 17.2.

Такое соединение архитектуры сервлетов с моделью Ring может быть выполнено двумя разными способами, в зависимости от выбранной стратегии развертывания:

1. Если вы собираетесь использовать один из встраиваемых серверов приложений (наиболее популярными из которых являются Jetty и Glassfish), вы можете создать обертку для сервлета во

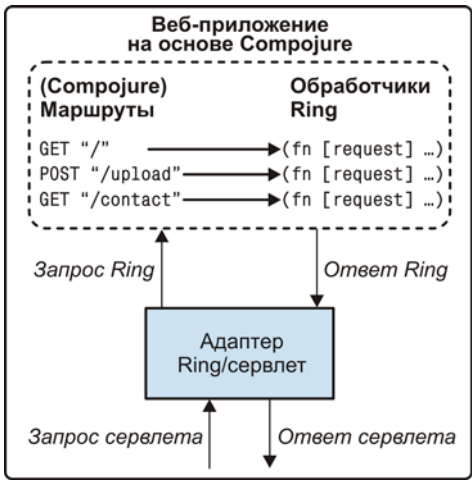

Рис. 17.2. Архитектура веб-приложения Clojure/Ring

время выполнения и просто передать ее серверу приложений, выполняющемуся в той же виртуальной машине JVM. Все необходимые операции производятся во время выполнения, благодаря чему отпадает необходимость в специальных стадиях упаковки и сборки, что особенно удобно во время разработки и тестирования в локальной системе.

2. При развертывании на автономных серверах приложений[1] обычно требуется упаковать веб-приложение в *.war*-файл. Этап упаковки легко добавляется в уже имеющуюся процедуру сборки и позволяет развертывать веб-приложения на Clojure на любых серверах приложений, придерживающихся стандартов (включая серверы приложений на таких платформах, как Google App Engine[2] и Amazon Elastic Beanstalk[3])[4].

---

[1] Существуют десятки зрелых, хорошо поддерживаемых серверов приложений, каждый из которых предоставляет собственный набор дополнительных особенностей помимо сервлетов и других стандартных механизмов JEE.

[2] https://github.com/gcv/appengine-magic.

[3] http://aws.amazon.com/elasticbeanstalk.

[4] Обратите внимание, что на платформе Heroku, ставшей одной из наиболее популярных платформ развертывания для Clojure в последнее время, применяется совершенно иной подход к развертыванию приложений; подробности см. в разделе «Clojure на платформе Heroku», в главе 20.

Оба способа имеют свои достоинства и недостатки Развертывание на отдельном сервере приложений предполагает настройку и выполнение дополнительных этапов сборки и упаковки. С другой стороны, переход на использование встраиваемого сервера приложений (иногда такие решения называют *бесконтейнерными* (containerless)) вынуждает создавать и поддерживать механизмы начального запуска, развертывания и управления, вместо того, чтобы просто пользоваться (обычно тщательно протестированными) средствами, поддерживаемыми отдельным сервером приложений. Мы рекомендуем использовать в первую очередь привычное решение, и обязательно поэкспериментировать с обоими, чтобы потом выбрать способ, лучше подходящий для вашего проекта и команды.

В главе 16 мы уже показывали, как запускать приложения из оболочки REPL во встраиваемой среде Jetty; теперь займемся упаковкой в *.war*-файл, что позволит нам развертывать приложения на серверах приложений и платформах промышленного уровня.

Упаковка веб-приложения

Веб-приложения на Java упаковываются в *.war*-файлы, расширенную версию *.jar*-файлов, обсуждавшихся в разделе «Артефакты и координаты», в главе 8[1]. Типичный *.war*-файл включает:

- ресурсы, такие как статические HTML-файлы и изображения;
- различные данные, помещаемые в каталог *WEB-INF*, в число которых входят:
  - файл *web.xml*, описывающий, как должен развертываться *.war*-файл на сервере веб-приложений;
  - подкаталог `lib`, где может находиться произвольное количество *.jar*-файлов; здесь обычно находятся промежуточные зависимости веб-приложения, что и делает *.war*-файлы автономными развертываемыми модулями (в противоположность другим архитектурам серверных приложений, требующих обширных настроек на стороне сервера, которые гарантировали бы доступность зависимостей развертываемых приложений);

---

[1] Мы лишь поверхностно познакомимся здесь с особенностями управления и развертывания веб-приложений, в частности с процедурой определения различных параметров в файле *web.xml*, включаемом в *.war*-архив. Более подробную информацию о *.war*-файлах и о параметрах настройки в *web.xml* можно получить в руководстве по адресу: http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WCC3.html.

- подкаталог *classes*, содержащий файлы с исходными текстами на языке Clojure, файлы JVM-классов (включая файлы классов, полученные в результате предварительной компиляции файлов с исходными текстами на языке Clojure) и другие ресурсы; то есть, в нем хранится содержимое типичного *.jar*-файла и «верхнеуровневый» программный код веб-приложения (в противоположность его зависимостям).

Основой соглашений о формировании *.war*-архивов является файл *web.xml*. Он определяет порядок развертывания приложения; в частности, с его помощью можно:

- ❑ «монтировать» отдельные сервлеты в конкретные каталоги, что дает возможность упаковать в один *.war*-файл несколько независимых приложений, размещаемых в разных каталогах;
- ❑ указывать, какие классы статических ресурсов (CSS, JavaScript и файлы изображений) должны обслуживаться сервером приложений непосредственно, без передачи их смонтированным сервлетам;
- ❑ настраивать поведение пользовательских сеансов, включая предельное время ожидания активности в сеансе и механизм хранения данных, который должен использоваться для сохранения информации о сеансе;
- ❑ настраивать службы и механизмы сервера приложений.

Ниже приводится простой пример содержимого файла *web.xml*, где определяется единственный сервлет (`com.clojurebook.hello_world`), смонтированный в корень (`/`) контекста сервера приложений, и перечисляет расширения файлов статических ресурсов, обслуживаемых сервлетом «по умолчанию» (каждый сервер приложений имеет такой сервлет, обслуживающий статические ресурсы):

Пример 17.1. Простой файл web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                               http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <servlet-name>app</servlet-name>
        <servlet-class>com.clojurebook.hello_world</servlet-class>
    </servlet>
    <servlet-mapping>
```

```
        <servlet-name>app</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>*.css</url-pattern>
        <url-pattern>*.js</url-pattern>
        <url-pattern>*.png</url-pattern>
        <url-pattern>*.jpg</url-pattern>
        <url-pattern>*.gif</url-pattern>
        <url-pattern>*.ico</url-pattern>
        <url-pattern>*.swf</url-pattern>
    </servlet-mapping>
</web-app>
```

Это – файл *web.xml*, который можно использовать в веб-приложении на Clojure, собираемом с помощью Maven. В Leiningen применяется иной подход, где файл *web.xml* генерируется исходя из конфигурации в файле *project.clj*. Как будет показано далее в этой главе, выбор инструмента сборки в действительности влияет лишь на незначительные детали, связанные с упаковкой и организацией веб-приложения.

Сборка .war-файлов с помощью Maven

Как было показано в разделе «Maven», в главе 8, по умолчанию инструмент сборки Maven создает *.jar*-файлы, по крайней мере, если в файле *pom.xml* определяется элемент `<packaging>` с содержимым `clojure` (или `jar`, по умолчанию). Если изменить содержимое элемента `<packaging>` на `war`, проект будет упаковываться в *.war*-файл командой `mvn package` (или любой другой фазой, зависящей от фазы `package`, такой как `install` или `deploy`). Так как мы не собираемся использовать тип упаковки `clojure`, нам также необходимо включить цель `compile` расширения `clojure-maven-plugin` в фазу `compile` в настройках Maven:

Пример 17.2. Файл pom.xml с настройками сборки простого проекта веб-приложения на Clojure

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.clojurebook</groupId>
```

```
<artifactId>sample-maven-web-project</artifactId>
<version>1.0.0</version>
<packaging>war</packaging>

<dependencies>
     <dependency>
          <groupId>org.clojure</groupId>
          <artifactId>clojure</artifactId>
          <version>1.3.0</version>
     </dependency>
     <dependency>
          <groupId>compojure</groupId>
          <artifactId>compojure</artifactId>
          <version>1.0.1</version>
     </dependency>
     <dependency>
          <groupId>ring</groupId>
          <artifactId>ring-servlet</artifactId>
          <version>1.0.1</version>
     </dependency>
</dependencies>

<build>
     <plugins>
          <plugin>
               <groupId>com.theoryinpractise</groupId>
               <artifactId>clojure-maven-plugin</artifactId>
               <version>1.3.8</version>
               <extensions>true</extensions>
               <configuration>
                    <warnOnReflection>true</warnOnReflection>
                    <temporaryOutputDirectory>
                         false
                    </temporaryOutputDirectory>
               </configuration>

               <executions>
                    <execution>
                         <id>compile-clojure</id>
                         <phase>compile</phase>
                         <goals>
                              <goal>compile</goal>
                         </goals>
                    </execution>
```

```
                </executions>
            </plugin>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>maven-jetty-plugin</artifactId>
                <version>6.1.15</version>
                <configuration>
                    <webAppConfig>
                        <extraClasspath>
                          src/main/webapp,src/main/resources,src/main/clojure
                        </extraClasspath>
                    </webAppConfig>
                    <reload>manual</reload>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

В этот пример файла *pom.xml* также включены зависимости для библиотек Compojure и Ring, и некоторые настройки расширения `maven-jetty-plugin`, позволяющие запускать веб-приложение локально, используя встраиваемый сервер приложений Jetty, идеально подходящий для разработки и тестирования на локальном компьютере.

Наше простое веб-приложение состоит из единственного обработчика. Поскольку при сборке с помощью Maven мы можем включить свой собственный файл *web.xml* и полностью переложить обслуживание запросов к статическим файлам на сервер приложений, нам не требуется определять дополнительные маршруты для обслуживания изображений и других подобных ресурсов:

Пример 17.3. com.clojurebook.hello-world

```
(ns com.clojurebook.hello-world
  (:use
    [ring.util.servlet :only (defservice)]
    [compojure.core :only (GET)])
  (:gen-class
    :extends javax.servlet.http.HttpServlet))

(defservice
  (GET "*" {:keys [uri]}
       (format "<html>
                 URL requested: %s
```

```
      <p>
        <a href=\"/wright_pond.jpg\">
          Image served by app server via web.xml <servlet-mapping>
        </a>
      </p>
      </html>"
    uri)))
```

Сборка .war-файлов с помощью Leiningen

`lein-ring`[1] – это расширение для Leiningen, поддерживающее массу возможностей, которые могут пригодиться при управлении проектом веб-приложения на основе библиотеки Ring с использованием Leiningen. Одной из таких возможностей является создание *.war*-файлов с включенными в их состав файлов *web.xml*, сгенерированными на основе конфигурационного файла проекта *project.clj*. Этот подход полностью отличается (к тому же он намного короче и проще!) от подхода, используемого в Maven, где требуется вручную создавать файл *web.xml*.

Пример 17.4. Файл project.clj с настройками простого веб-приложения на Clojure

```
(defproject com.clojurebook/sample-lein-web-project "1.0.0"
  :dependencies [[org.clojure/clojure "1.4.0"]
                 [compojure/compojure "1.0.1"]
                 [ring/ring-servlet "1.0.1"]]
  plugins [[lein-ring "0.6.2"]]
  :ring {:handler com.clojurebook.hello-world/routes})
```

Единственная настройка, необходимая расширению `lein-ring`, – слот `:ring :handler`, где указывается полное квалифицированное имя переменной, хранящей обработчик запросов верхнего уровня. На основе этого имени расширение `lein-ring` сгенерирует класс сервлета (в данном случае с именем `com.clojurebook.servlet`), который будет передавать запросы нашему обработчику, и файл *web.xml*, определяющий сервлет и монтирующий его в корневой путь URL (/). Это эквивалентно созданию файла *web.xml* вручную, использованию `gen-class` и `defservice` для создания класса сервлета, и делегированию обработки запросов в проекте на основе Maven.

---

[1] Полное описание можно найти по адресу: https://github.com/weavejester/lein-ring.

Если теперь, имея такой файл *project.clj*, выполнить команду `lein ring uberwar`, она создаст *.war*-файл, который можно использовать для развертывания приложения на любом сервере приложений.

Поскольку файл *web.xml*, сгенерированный расширением `lein-ring`, не определяет, какие статические ресурсы должны обрабатываться сервлетом по умолчанию, в главный обработчик необходимо включить маршрут, который будет соответствовать этим ресурсам, находящимся в пути поиска classpath:

Пример 17.5. com/clojurebook/hello_world.clj

```
(ns com.clojurebook.hello-world
  (:use
    [compojure.core :only (GET defroutes)]
    [compojure.route :only (resources)]))

(defroutes routes
  (resources "/")
  (GET "*" {:keys [uri]}
       (format "<html>
                  URL requested: %s
                  <p>
                    <a href=\"/wright_pond.jpg\">
                      Image served by compojure.route/resources
                    </a>
                  </p>
                  </html>"
               uri)))
```

Внимание. Расширение `lein-ring` дает возможность определять лишь ограниченное подмножество параметров настройки, допустимых в файлах web.xml, и в настоящее время не позволяет предоставлять собственные файлы web.xml. Поэтому, если вам потребуется использовать некоторые особенности спецификации сервлетов (такие как сервлет-фильтры, параметры контекста и карту маршрутов сервлета по умолчанию, подобную той, что была показана в примере 17.1, чтобы переложить обработку статических ресурсов на сервер приложений), вам необходимо будет:

- воспользоваться расширением `leiningen-war`[1] (который только собирает .war-файлы и не поддерживает возможность развертывания на локальном сервере приложений Jetty, как расширение `lein-ring`) или;
- добавить сценарии, выполняющие дополнительные операции после вызова расширения `lein-ring`, чтобы заменить автоматически сгенерированный файл web.xml собственной версией, или;
- использовать Maven.

[1] https://github.com/alienscience/leiningen-war.

Запуск веб-приложений на локальном компьютере

Независимо от используемого инструмента сборки, Maven или Leiningen, во время разработки и тестирования вы можете пользоваться встраиваемым сервером приложений Jetty для запуска веб-приложения на локальном компьютере. Это позволяет быстро разрабатывать новые особенности и вносить исправления, без необходимости выполнять полный цикл упаковки и развертывания, как того требует эксплуатационная среда или удаленная среда тестирования.

Maven. Как отмечалось выше, в разделе «Сборка .war-файлов с помощью Maven», наш файл *pom.xml* включает настройки для расширения `maven-jetty-plugin`. Если внутри проекта с таким конфигурационным файлом вызвать команду `mvn jetty:run`, она запустит Jetty на `localhost` с портом 8080 и выполнит веб-приложение проекта. По мере внесения изменений в исходный код на Clojure, статические ресурсы или файл *web.xml*, желательно, чтобы эти изменения немедленно отражались на поведении веб-приложения, выполняющегося под управлением Jetty, без полной его остановки и перезапуска. Для этого нажмите клавишу **Return** в окне консоли, где запущен Maven, в результате чего будет выполнена перезагрузка контекста веб-приложения в Jetty, которая выполняется намного быстрее, чем последовательность операций остановки процесса Maven/Jetty и повторного его запуска.

Leiningen. Поддержка возможности запуска веб-приложения на локальном компьютере в Leiningen почти не отличается от аналогичной в Maven. Команда `lein ring server` запустит сервер Jetty, который будет передавать все запросы корневому обработчику, указанному в слоте `:ring :handler`, в файле *project.clj*. Расширение `lein-ring` реализует несколько иной подход к загрузке изменений в действующее веб-приложение: вместо ожидания явной команды на перезагрузку контекста приложения, `lein-ring` перезагружает все файлы с исходными текстами на Clojure с помощью функции `require` всякий раз, когда принимает запрос.

Загрузка измененного кода через «удаленный» сеанс REPL. Более гибкое решение, чем могут предложить `maven-jetty-plugin` и `lein-ring`, заключается в том, чтобы связать сервер REPL с веб-приложением и настроить его запуск после развертывания. это по-

зволит подключаться к «удаленному» серверу REPL из среды разработки (такой как Eclipse + Counterclockwise или Emacs + SLIME) и загружать код Clojure, когда это потребуется, вместо того, чтобы полагаться на предельное время ожидания, выполнять переключение контекста в терминале, где был запущен Jetty, или откладывать сохранение изменений на диск, пока вы не будете готовы загрузить эти изменения в действующее приложение.

Этот прием можно применять не только к веб-приложениям и не только на локальном компьютере; подробности см. в главе 10.

Развертывание веб-приложения

Возможность упаковывать веб-приложения на Clojure в стандартные *.war*-файлы дает нам немалую гибкость: все средства и приемы развертывания веб-приложений на Java с тем же успехом могут применяться к веб-приложениям на Clojure. В общем случае для развертывания приложения требуется следующее:

1. Установить и настроить сервер приложений.
2. Скопировать на сервер *.war*-файл, созданный в процессе сборки.
3. Перезапустить сервер приложений, если необходимо.
4. При необходимости вернуть предыдущую версию *.war*-файла (если, например, последняя версия содержит регрессии).

Конечно, эти операции можно выполнять вручную или каким-то иным путем – программисты и системные администраторы *давно* выработали «практические» приемы развертывания приложений. Но, если в вашей организации уже существует процедура развертывания веб-приложений на Java, вы почти наверняка сможете использовать ее для развертывания своих новых, сверкающих веб-приложений на Clojure.

Однако при желании можно использовать специализированные инструменты для Clojure, которые делают развертывание приложений намного проще, удобнее и позволяют автоматизировать этот процесс намного полнее. Далее мы рассмотрим один из них – службу Amazon Elastic Beanstalk – нашедший широкое применение в веб-приложениях на Clojure, автоматизирующий подготовку и настройку серверов, и выполняющий развертывание приложений на этих серверах.

Развертывание приложений на Clojure с помощью Amazon Elastic Beanstalk

Amazon Elastic Beanstalk (EB) – это *платформа как услуга* (platform as a service), предоставляющая инструменты управления автоматизацией и развертыванием, основанные на службах Amazon Web Services (AWS) EC2 низкоуровневых вычислений и распределения нагрузки. Служба EB позволяет программно управлять *окружениями* (коллекциями из одного или нескольких серверов приложений, находящихся за механизмом распределения нагрузки (load balancer)), куда можно разворачивать различные версии своего приложения.

Механизмы распределения нагрузки (load balancers), используемые в EB, интегрированы в общий механизм обеспечения, поэтому, когда приложение испытывает высокую нагрузку (по определяемым вами параметрам, таким как количество запросов или совокупный объем трафика в минуту), соответствующее окружение EB расширяется за счет увеличения количества серверов приложений, обслуживающих эту нагрузку. В частности, если ваше приложение использует средства AWS хранения данных[1] или некоторую другую базу данных, которая может размещаться в AWS[2], EB может оказаться весьма привлекательным способом развертывания приложений, способным обслуживать полный цикл разработки, развертывания и сопровождения приложения.

Инфраструктура AWS предоставляет исчерпывающий Java API для своих служб, включая EB, поэтому взаимодействия с ней из программ на языке Clojure реализуются достаточно просто. Кроме того, существует расширение lein-beanstalk для Leiningen, обеспечивающее возможность интеграции этого инструмента с EB[3].

Базовые настройки и развертывание. Чтобы подготовить приложение к использованию с расширением `lein-beanstalk`, необходимо внести пару изменений в файл проекта веб-приложения, управляе-

---

[1] AWS поддерживает три службы «управляемых» баз данных, которые хорошо сочетаются с минимальными накладными расходами Elastic Beanstalk: DynamoDB и SimpleDB – нереляционные хранилища данных (последнее является предшественником первого), – и AmazonRDS, обеспечивающую доступ к реляционным базам данных Oracle и MySQL. Один из авторов книги занимается поддержкой реализации Clojure API для SimpleDB, доступной по адресу: https://github.com/cemerick/rummage.

[2] Такую как Cloudant на базе кластеров CouchDB: https://cloudant.com.

[3] https://github.com/weavejester/lein-beanstalk.

мого с помощью Leiningen. Во-первых, в файле *project.clj* необходимо добавить расширение `lein-beanstalk` в вектор `plugins`.

```
[lein-beanstalk "0.2.2"]
```

Во-вторых, необходимо добавить маршрут Compojure для поддержки «мониторинга» со стороны Elastic Beanstalk. Вашему развернутому приложению регулярно будет посылаться `HEAD`-запрос с корневым (/) URI. Если приложение не ответит на этот запрос, Elastic Beanstalk будет полагать, что приложение обрушилось и попытается развернуть его и запустить повторно. Для этого примера используем службу сокращения URL, созданную в разделе «Маршрутизация запросов с помощью Compojure», в главе 16. Настроим маршрут мониторинга для Beanstalk в отдельном пространстве имен, и – как еще одно подтверждение возможности комбинирования маршрутов Compojure и обработчиков Ring – просто добавим на верхний уровень маршрут к фактической службе сокращения URL:

Пример 17.6. com.clojurebook.url-shortener.beanstalk

```
(ns com.clojurebook.url-shortener.beanstalk
  (:use [compojure.core :only (HEAD defroutes)])
  (:require [com.clojurebook.url-shortener :as shortener]
            [compojure.core :as compojure]))

(compojure/defroutes app
  ; Этот маршрут обработки HEAD-запросов необходим для поддержки мониторинга
  ; со стороны службы Amazon Elastic Beanstalk, определяющей
  ; работоспособность приложения по успешным ответам на запросы
  ; HEAD, отправляемые по адресу /
  (compojure/HEAD "/" [] "")
  shortener/app)
```

Последнее, изменение, которое осталось внести в файл проекта, – добавить настройку `:ring :handler`, потому что для создания *.war*-файлов расширение `lein-beanstalk` использует `lein-ring`. Ниже показано содержимое файла *project.clj* проекта службы сокращений URL с подключенным расширением `lein-beanstalk` и соответствующим значением в слоте `:ring :handler`:

Пример 17.7. project.clj

```
(defproject com.clojurebook/url-shortener "1.0.0"
  :description
```

```
    "A toy URL shortener HTTP service written using Ring and Compojure."
  :dependencies [[org.clojure/clojure "1.3.0"]
                 [compojure "1.0.1"]
                 [ring "1.0.1"]]
  plugins [[lein-beanstalk "0.2.2"]]
  :ring {:handler com.clojurebook.url-shortener.beanstalk/app})
```

В заключение необходимо добавить в файл *~/.lein/init.clj* параметры аутентификации в AWS[1]; они будут использоваться для подтверждения ваших прав на выполнение всех операций с помощью `lein-beanstalk`:

Пример 17.8. ~/.lein/init.clj

```
(def lein-beanstalk-credentials
  {:access-key "XXXXXXXXXXXXXX"
   :secret-key "YYYYYYYYYYYYYY"})
```

После внесения всех этих изменений можно попробовать развернуть приложение в Elastic Beanstalk. Для этого достаточно выполнить единственную команду:

```
lein beanstalk deploy development
```

Эта команда потребует от `lein-beanstalk`:

1. Создать *.war*-файл веб-приложения с помощью `lein-ring`.
2. Передать этот файл службе Amazon S3.
3. Создать приложение для Elastic Beanstalk с именем, совпадающим с именем проекта, если оно еще не существует[2].
4. Создать окружение с именем `development` для приложения Elastic Beanstalk.

---

[1] Параметры аутентификации никогда не должны сохраняться в файле *project.clj* или в любом другом файле в вашем проекте, который может быть сохранен в системе управления версиями. Последствия могут оказаться самыми неприятными, если эта информация попадет в общедоступные репозитории, такие как GitHub или BitBucket.

[2] Например, в нашем примере проекта `lein-beanstalk`, с координатами `com.clojurebook/url-shortener`, приложение получит имя `url-shortener`. Имена приложений Elastic Beanstalk должны быть глобально уникальными, поэтому, когда вы будете опробовать (любой) пример, не забудьте изменить идентификатор артефакта (`artifactID`) на любое другое уникальное значение.

5. Проверить, был ли развернут *.war*-файл, переданный службе S3, в окружении `development`.

Когда команда `deploy` завершится (что может занять несколько минут, особенно когда развертывание данного приложения или в данном окружении выполняется впервые), расширение `lein-beanstalk` сообщит, что приложение запущено и готово к работе[1].

Управление версиями приложения. Служба Elastic Beanstalk сохраняет все предыдущие версии приложения в S3, поэтому в любой момент можно вернуться к любой из них с помощью консоли AWS. Для создания версий в EB, `lein-beanstalk` использует номера версий, указанные в файле *project.clj*, поэтому вы легко сможете понять, какая версия развернута и как она связана с предыдущими версиями приложения. Узнать самую последнюю версию, выгруженную в Elastic Beanstalk, можно с помощью команды `lein beanstalk info`:

```
% lein beanstalk info
Application Name: url-shortener
Last 5 Versions:  1.0.0-SNAPSHOT-20111219051007
                  1.0.0-SNAPSHOT-20111219045316
Created On:       Mon Dec 19 04:53:53 EST 2011
Updated On:       Mon Dec 19 04:53:53 EST 2011
Deployed Envs:    development (Ready)
```

Убедившись, что развернутая версия устраивает вас, вы можете с помощью команды `lein beanstalk clean` удалить из S3 все неиспользуемые версии и соответствующие им *.war*-файлы.

Окружения. Вы можете создать в Elastic Beanstalk любое количество окружений для одного и того же приложения, с любыми именами по вашему выбору. По умолчанию расширение `lein-beanstalk` определяет три окружения: `development`, `staging` и `production`, первое из которых было использовано в нашем примере выше. Вы можете управлять этими умолчаниями в своем файле *project.clj*. За дополнительной информацией обращайтесь к документации для `lein-beanstalk`.

Получить различные сведения о каждом окружении можно с помощью команды `lein beanstalk info <имя-окружения>`:

---

[1] ...по адресу URL, имеющему вид: http://*имя-проекта*.elasticbeanstalk.com. Вы можете настроить запись `CNAME` в DNS своего домена, чтобы связать доменное имя, например, www.yourdomain.com, с этим URL в домене elasticbeanstalk.com, и обеспечить прозрачное обслуживание на фирменных сайтах Elastic Beanstalk.

```
% lein beanstalk info development
Environment ID:     e-cnjm4hrqki
Application Name:   url-shortener
Environment Name:   development
Description:
URL:                url-shortener-dev.elasticbeanstalk.com
Load Balancer URL:  awseb-development-1574221210.us-east-1.elb.amazonaws.com
Status:             Ready
Health:             Green
Current Version:    1.0.0-SNAPSHOT-20111219051007
Solution Stack:     32bit Amazon Linux running Tomcat 6
Created On:         Mon Dec 19 05:10:44 EST 2011
Updated On:         Mon Dec 19 05:13:11 EST 2011
```

За пределами развертывания простых веб-приложений

Конечно, в зависимости от требований, может потребоваться знать больше, намного больше о развертывании приложений, чем мы смогли рассказать здесь. Многие проекты (даже очень большие и высоконагруженные сайты) прекрасно могут работать под управлением таких служб как Elastic Beanstalk. Однако у кого-то может возникнуть необходимость автоматизировать процессы развертывания, не ограничивающиеся простым копированием *.war*-файлов в действующий контейнер. Точное управление подготовкой и настройкой веб-интерфейсов и механизмов распределения нагрузки, полный контроль над средствами поддержки баз данных, нестандартные настройки маршрутизации и мониторинга сети, возможно, гетерогенное сочетание облачных служб и внутренней инфраструктуры... Любая из этих проблем может затруднить то, что иначе, было бы лишь вопросом применения единственной и простой в использовании облачной службы, и потребовать применения дополнительных инструментов, помимо тех, что используются в общем случае. Pallet – это широко используемый комплект инструментов для Clojure, решающий подобные проблемы; см. раздел «Pallet» в главе 20.

Какие бы инструменты вы ни использовали, и какие потребности ни испытывали, никогда не забывайте, что решения, применимые к Java-приложениям, в равной степени пригодны и для Clojure-приложений. Точно так же вы всегда можете использовать инструменты, изначально не предназначавшиеся для Clojure (такие как Chef, Puppet и другие), для развертывания приложений на Clojure и управления ими, просто используя приемы и идиомы, предусматриваемые этими инструментами для приложений на Java и JVM в целом.

Часть V
РАЗНОЕ

Глава 18. Выбор форм определения типов

В языке Clojure имеется множество различных форм определения типов:

- `deftype`, `defrecord` и `reify` – основные абстракции определения типов в языке Clojure, подробно рассматривавшиеся в разделе «Определение собственных типов», в главе 6;
- ассоциативные массивы всех видов, особенно удобные для создания наиболее гибких фактических типов, обсуждались в главе 3;
- `proxy` и `gen-class`, основное назначение которых – обеспечить полноценную возможность обмена данными с Java и JVM, были покрыты в разделе «Определение классов и реализация интерфейсов», в главе 9.

Каждая из этих форм имеет свои достоинства и недостатки. Особенно начинающим осваивать Clojure может быть сложно определить, какую форму использовать для определения того или иного типа. Когда следует использовать `deftype` вместо `defrecord`, `gen-class` вместо `deftype` или `proxy` вместо `reify`?

В разделах, упомянутых выше, мы постарались исследовать все нюансы этих форм. Однако иногда полезно иметь зрительный ориентир, даже самого общего характера. Учитывая это, мы надеемся, что вы найдете полезной диаграмму на рис. 18.1. Начиная с момента, когда у вас появится потребность определить новый тип данных на языке Clojure, она поможет вам в наиболее важных пунктах сделать выбор между разными формами определения типов, чтобы в конечном итоге вы могли прийти к наиболее подходящей в вашей ситуации[1]:

---

[1] Каноническую и более свежую версию этой диаграммы всегда можно найти по адресу: https://github.com/cemerick/clojure-type-selection-flowchart, а также переводы ее на разные языки, включая голландский, немецкий, японский, португальский и испанский.

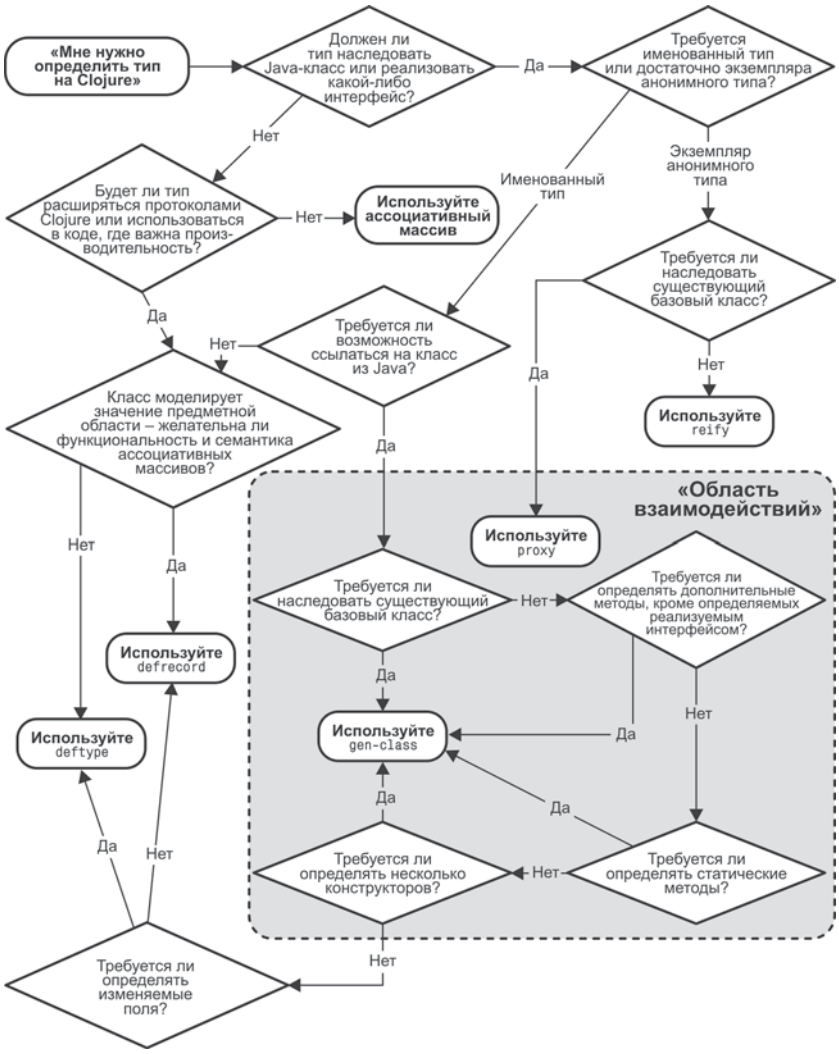

Рис. 18.1. Выбор формы определения типа

Прямоугольник, помеченный надписью «Область взаимодействий», соответствует случаям (например, когда необходимо определить несколько конструкторов) и формам (`proxy` и `gen-class`), относящимся исключительно к области поддержки взаимодействий

между Clojure и JVM. Эти формы прекрасно справляются со своими задачами, но знайте, что используя их, вы выходите за рамки «родных» для Clojure абстракций. Более простые формы определения типов могут лучше соответствовать вашим потребностям, если только вы не определяете новый тип исключительно для организации взаимодействий.

Примером «кода, где важна производительность» может служить доступ к слотам в глубоком цикле. В такой ситуации:

- использование обычных ассоциативных массивов может оказаться убийственным;
- использование обращений вида `:keyword` при работе с записями или экземплярами типов, объявленных с помощью `deftype`, позволит довольно близко приблизиться к производительности Java;
- использование прямых обращений к полям (например, `(.field val)`) при работе с записями или экземплярами типов, объявленных с помощью `deftype`, позволить добиться производительности, равной производительности кода на Java.

Это не означает, что непосредственный доступ к полям должен использоваться повсеместно или почти повсеместно. Это один из приемов оптимизации и его следует использовать, только когда это действительно необходимо, особенно учитывая недостатки такого подхода: за высокую эффективность непосредственного доступа к полям придется заплатить тесной зависимостью кода от конкретного типа, что зачастую осложняет реализацию универсальных функций и ограничивает возможности их композиции[1].

---

[1] Не забывайте, что «преждевременная оптимизация – корень всех зол», как сказал профессор Кнут (Knuth).

Глава 19. Внедрение Clojure

(или, как протащить Clojure за спиной у начальника[1])

Грустно, но факт, что многие программисты, если не большинство, в своей повседневной работе используют языки и инструменты, которые вызывают у них недовольство. По историческим ли причинам, из-за организационной инерции или жестких производственных требований, мы часто оказываемся ограничены в своих желаниях использовать *что-то* другое для выполнения своей работы.

Такое положение вещей может доставлять особенные огорчения, если вы достаточно далеко продвинулись в понимании и оценке возможностей Clojure, чтобы появилось желание использовать его в своей работе. В этой главе мы хотим дать вам краткое руководство, шпаргалку, ряд убедительных аргументов и стратегий, чтобы помочь вам успешно внедрить Clojure в свою работу. При этом мы надеемся, что ваши дни станут более производительными, ночи менее беспокойными, а бизнес более выгодным.

Только факты...

Clojure – новейший и инновационный язык. За свою короткую историю Clojure превратился в широко известный язык программирования. Еще пять лет назад казалось невероятным, что такую популярность сможет получить язык, поощряющий функциональный стиль программирования, по умолчанию использующий неизменяемые структуры данных, поддерживающий легко управляемые примитивы конкуренции и параллелизма, предлагающий обширные средства метапрограммирования, выполняющийся под управлением JVM и при этом почти не уступающий в производительности Java.

Об этом можно долго говорить, и есть все основания полагать, что Clojure продолжит расширять свои границы и осваивать новые

---

[1] Шутка, но... в каждой шутке есть доля истины!

территории, и то, что он уже предлагает, продолжит зреть и находить все более широкое понимание.

Clojure опробован, протестирован и имеет надежную основу. Clojure имеет глубокие корни, хотя и является достаточно новым языком программирования. Рич Хикки (Rich Hickey), создатель языка Clojure, разработал не менее трех проектов до создания Clojure, в основе которых лежала организация взаимодействий между средой Lisp и JVM и/или .NET[1]. Эти эксперименты проложили путь к созданию Clojure, который достиг и утвердил определенный баланс между основными особенностями языка Lisp и средой VM.

Конечно, Clojure создавался под значительным влиянием прежних диалектов Lisp, а также многих современных языков (включая Java, Python, Ruby, JavaScript). Наиболее привлекательными чертами, которые унаследовал Clojure, являются: полноценная поддержка функционального программирования, макросов, средств метапрограммирования, которые, хотя и насчитывают не одно десятилетие практического применения до появления Clojure, но остаются уникальными за пределами семейства языков Lisp.

Во многих отношениях нацеленность Clojure на JVM замыкает круг, который был начат, когда создатели Java JVM заимствовали огромное количество особенностей ранних, таких дружелюбных и продуктивных систем программирования на Lisp[2].

Clojure позволяет использовать все преимущества JVM. Многие программисты, привлеченные выразительностью и относительной гибкостью интерпретируемых языков (таких как Ruby и Python) долгое время жаждали получить более высокопроизводительное окружение. Специально для тех, кто предпочитает пользоваться возможностью динамической типизации, Clojure предоставляет самое лучшее из двух миров: высочайшую выразительность и опору на JVM, которая за почти 20-летнюю историю выросла в невероятно обширную, многоцелевую вычислительную платформу.

---

[1] В хронологическом порядке: DotLisp, работающий на платформе .NET (http://dotlisp.sourceforge.net); jFli (http://jfli.sourceforge.net) – реализация для JVM; и FOIL (http://foil.sourceforge.net), способный работать с любой виртуальной машиной.

[2] Гай Стил (Guy L. Steele), один из создателей оригинального языка программирования Scheme и автор книг «Java Language Specification» и «Common Lisp, the Language» (ставшей основой стандарта ANSI Common Lisp), широко известен как автор слов: «Нам удалось перехватить массу [программистов на C++] на полпути к Lisp.».

Опора на JVM означает, возможность пользоваться всеми плодами широкой распространенности этой платформы, такими как: надежность и высокая производительность, обширная экосистема высококачественных библиотек сторонних разработчиков (как открытых, так и коммерческих), зрелая поддержка серверных технологий и инструментальное обеспечение, согласованная система управления зависимостями, и так далее.

Clojure позволяет использовать прежние наработки на Java. Если в настоящее время вы используете Java или имеете другие инвестиции в JVM, Clojure поможет вам использовать их. Веб-приложения на Clojure прекрасно уживаются с веб-приложениями на Java, Java-библиотеки можно вызывать из программ на Clojure, и точно так же можно вызывать функции Clojure и создавать экземпляры типов Clojure из программ на Java (или на любом другом языке, выполняющемся под управлением JVM, включая JRuby, Jython, JavaScript [через Rhino], Scala, Groovy, и так далее). Все, что вы узнали о сборке, упаковке, непрерывной интеграции, об особенностях работы JVM и ее настройке, в равной степени применимо и к Clojure.

Clojure – это прибавка к вашим инвестициям в JVM, а не радикальный уход в сторону.

«Clojure – всего лишь .jar-файл». Как следствие из того, что Clojure позволяет использовать прежние наработки на Java, реализация этого языка в действительности является «всего лишь *.jar*-файлом». Это означает, что его можно упаковать в дистрибутив приложения, как одну из зависимостей, вместе с исходными текстами на языке Clojure (или файлами классов, полученными в ходе предварительной компиляции исходных текстов), и ваши клиенты и заказчики ни о чем не догадаются.

Мы не предлагаем отвергать требования или обманывать клиентов. Однако, если у вас есть право и свобода выбора инструментов для создания программного обеспечения, тогда использование Clojure, как части общего решения, мало отличается от выбора грамматики для ANTLR, библиотеки Lucene для организации хранилища данных, или использования Visual Studio для создания настольных приложений на Java для Windows.

Подчеркните особую продуктивность

Меньше кода, больше пользы. Велика вероятность, что решение любой проблемы на Clojure будет более кратким, менее сложным

и более выразительным, чем решение той же проблемы на другом языке. В программах на Clojure *намного* меньше изменяемых данных, с которыми придется сражаться. Механизмы конкуренции и параллелизма имеют простую и последовательную семантику, они устраняют потребность в блокировках и не подвержены взаимоблокировкам. Классы и типы являются необязательными инструментами, а методы чтения/записи вообще не нужны. Синтаксические шаблоны могут быть абстрагированы с помощью макросов. Этот список можно продолжать и продолжать.

Использование Clojure наверняка сможет поднять вашу личную продуктивность так же, как ее смог поднять язык Java, когда вы еще использовали C/C++, или так же, как смогли это Ruby или Python, когда вы еще использовали Java. Эти преимущества распространятся на всю вашу команду, как только исчезнет влияние концептуальных и архитектурных недостатков, свойственных другим языкам.

Программисты всегда стремились к языкам высокого уровня и Clojure – это лишь еще один шаг в данном направлении.

Clojure позволяет применять инновационные решения для сложных проблем. Многое из того, о чем говорилось выше, имеет отношение к способности Clojure сохранить инвестиции, отвечать языковым предпочтениям, и так далее. Но еще одним из наиболее ощутимых преимуществ Clojure является способность помочь вам выделиться на общем фоне.

Большинство языков программирования навязывает определенные архитектурные решения, в которые вы должны втискивать свою предметную модель. В Clojure используется совершенно иной подход; он способствует использованию функционального стиля программирования и моделированию решений в терминах специализированных абстракций (предоставляя для этого такие инструменты, как протоколы, записи и мультиметоды). При этом Clojure дает возможность «платить» только за те возможности языка, которые вы используете, где «валютой» являются не наличные деньги, а сложность моделирования и мыслительные нагрузки.

Все это означает, что проектируя API библиотеки или целой системы на языке Clojure, вы не стеснены решениями, которые Clojure принимает за вас. Это может сделать решение мелких проблем экстраординарно простым, и возможным – решение очень сложных проблем.

Clojure способен расширить ваши возможности. Приходилось ли вам использовать инструменты, дающие вам или вашей организации непревзойденные преимущества перед конкурентами[1]?

Представьте, что получилось бы, если бы в 1999 году у вас была возможность использовать Ruby и Rails для разработки веб-приложений? Или, если бы вы могли создавать крупномасштабные системы с использование версий Java, появившихся в 2000 годах, в 1990? Подозреваем, что вы не упустили бы такую возможность.

В настоящий момент весь мир крутится на Java, PHP и C/C++, с незначительной примесью «экспериментальных» языков, таких Ruby, Python и Scala. Можно предположить, что Clojure никогда не получит такое же распространение, как Java, PHP или C++, но мы смело можем утверждать, что некоторые из его ключевых особенностей, такие как программная транзакционная память и неизменяемые структуры данных, уже заимствуются, адаптируются и иногда используются непосредственно другими, весьма восприимчивыми языками программирования. Однако, так же как Java не стала диалектом Lisp из-за того, что создатели JVM заимствовали множество приемов и особенностей из Lisp, другие языки, заимствующие некоторые особенности из Clojure, не станут равными ему по своим возможностям.

Если вы начнете использовать Clojure сегодня, вы получите непревзойденные преимущества на годы вперед.

Подчеркните широту сообщества

Clojure – открытый проект, участие в котором может принять любой желающий. Clojure распространяется с открытыми исходными текстами и на условиях свободной лицензии[2], позволяющей включать его в коммерческие продукты и использовать в коммерческих организациях (в дополнение к возможности использовать его в некоммерческих, благотворительных или личных целях).

---

[1] С нашей стороны было бы бестактностью не упомянуть здесь очерк Пола Грэма (Paul Graham) «Beating the Averages» (Побеждая посредственность), который как нельзя лучше подходит к обсуждению этой темы: http://www.paulgraham.com/avg.html. (перевод на русский язык: http://www.nestor.minsk.by/sr/2003/07/30710.html. – *Прим. перев.*)

[2] «Eclipse Public License», разрешающей коммерческое использование и распространение: http://www.eclipse.org/legal/epl-v10.html.

Хотя сам язык Clojure является отражением взглядов одного человека[1], вокруг проекта Clojure сплотилось весьма обширное сообщество: сотни отдельных участников (многие из которых представляют компании) зарегистрированы официально[2] и помогают развивать различные стороны проекта. На момент написания этих строк свой вклад в развитие самого языка Clojure внесли более 70 отдельных разработчиков.

Таким образом:

1. Использование Clojure не накладывает никаких юридических ограничений.
2. Если вы найдете ошибку в реализации Clojure, вы сможете помочь в ее устранении – либо отправив отчет об ошибке одному из зарегистрированных разработчиков, либо исправив ее самостоятельно, если предпочтете зарегистрироваться.

Clojure окружен большим, развивающимся и дружелюбным сообществом. Сообщество Clojure, и так достаточно обширное, продолжает разрастаться быстрыми темпами:

- главный список рассылки (http://groups.google.com/group/clojure) насчитывает более 6000 активных подписчиков;
- главный канал IRC[3] насчитывает более 400 активных респондентов;
- проекты разработки основных библиотек и инструментов Clojure имеют собственные списки рассылки (большинство из них находятся на Google Groups) и сопровождаются авторами и ведущими разработчиками;
- к моменту, когда вы будете читать эти строки, будет напечатано не менее шести книг о Clojure всеми основными издательствами.

Однако размер – это не главное. Некоторые языки программирования и фреймворки также имеют свои сообщества людей, эгоистично настроенных, недружелюбных и злых на язык. В отличие

---

[1] Рич Хикки (Rich Hickey) сыграл в Clojure ту же роль, что и Гвидо ван Россум (Guido Van Rossum) в Python, Юкихиро Мацумото (Yukihiro Matsumoto) в Ruby, Ларри Уолл (Larry Wall) в Perl и Бьерн Страуструп (Bjarne Stroustrup) в C++.

[2] Список всех официально зарегистрированных разработчиков можно найти на странице http://clojure.org/contributing.

[3] irc://irc.freenode.net/clojure или для подключения из веб-браузера: http://webchat.freenode.net/?channels=#clojure.

от них сообщество Clojure заработало репутацию дружелюбного и приветливого, по отношению к начинающим. Фактически, когда в 2011 году проводился опрос среди членов сообщества Clojure[1], на вопрос: «какие факторы оказали наиболее отрицательное влияние на использование Clojure», – лишь 2 процента респондентов (самое меньшее количество голосов среди других ответов на этот вопрос) ответили: «недружелюбная реакция сообщества».

Все это означает, что если вы захотите пообщаться с опытными программистами на Clojure – предложить или попросить помощь – вам будет куда прийти.

Clojure широко используется. Основная команда разработчиков Clojure поддерживает вики-страницу, где любая организация, использующая Clojure, может заявить о себе:

http://dev.clojure.org/display/community/Clojure+Success+Stories

Посетив эту страницу, вы узнаете, что: язык Clojure используют всемирно известные корпорации, такие как Citicorp и Akamai; молодые компании, такие как Backtype (ныне часть компании Twitter), The Climate Corporation (прежде носившая название Weatherbill) и Woven; консалтинговые компании, такие как Relevance; и научные организации, такие как институт молекулярной биомедицины общества Макса Планка (Max Planck Institute for Molecular Biomedicine).

Если ваша организация начнет использовать Clojure, она окажется в хорошей компании.

Будьте благоразумны

Не пытайтесь заткнуть круглое отверстие квадратной пробкой. Как и любой другой инструмент, Clojure обладает массой возможностей, которые особенно ярко проявляются, когда применяются там, где они могут принести наибольшую пользу. Если вы любите Clojure и хотите использовать его регулярно, прежде чем пытаться продвигать его в своей организации, вы должны убедиться, что он хорошо подходит для решения ваших задач.

---

[1] Этот опрос вот уже в течение двух лет проводится одним из авторов данной книги, чтобы иметь представление о настроениях и приоритетах в сообществе Clojure; полные результаты опроса доступны по адресу: http://cemerick.com/2011/07/11/results-of-the-2011-state-of-clojure-survey/.

То есть, если ваша организация занимается разработкой встраиваемых систем на C++ (к примеру), вы едва ли сможете убедить своих коллег переписать все, что было наработано, на языке Clojure[1].

Начинайте с малого и не торопитесь. Иногда постепенное движение вперед – лучший способ привнести что-то новое в существующую систему. Учитывая это, лучше и проще будет начать с создания тестов на Clojure для всех ваших проектов или с разработки внутренних и второстепенных инструментов. Постепенное формирование положительной репутации Clojure среди коллег часто оказывается более убедительной демонстрацией его возможностей и преимуществ, чем любая, самая лучшая презентация.

---

[1] По крайней мере не сразу и не сейчас. Однако в настоящее время ведутся исследования по созданию компиляторов с языка Clojure на низкоуровневые языки, которые можно было бы использовать для создания приложений для таких систем. Например: http://nakkaya.com/2011/06/29/ferret-an-experimental-clojure-compiler/.

Глава 20. Что дальше?

Прочитав эту книгу (или из желания заглянуть в будущее, чтобы узнать, что вас ждет, как только вы достаточно уверенно овладеете основами Clojure), вам, возможно, будет интересно познакомиться с проектами и ресурсами, перечисленными в этой главе. Это одни из самых интересных объектов во вселенной Clojure, известных нам. Одни помогут узнать Clojure еще лучше, другие могут пригодиться в «реальных» проектах, а третьи – просто достаточно увлекательные, чтобы взглянуть на них хотя бы раз.

Наконец, из-за того, что Clojure – относительно новый язык, окруженный бурно разрастающимся и развивающимся сообществом, новые удивительные разработки появляются как грибы после дождя. Если вы постоянно будете заглядывать на сайт http://clojurebook.com, мы приложим все усилия, чтобы удовлетворить ваше любопытство и познакомить вас с лучшими из лучших.

(dissoc Clojure 'JVM)

Существует еще две поддерживаемые реализации Clojure для других сред выполнения. Если вам понравилось все, что вы увидели в этой книге, но по каким-то причинам вынуждены использовать иную среду выполнения, отличную от JVM, тогда вам, возможно, стоит обратить внимание на ClojureCLR или ClojureScript.

ClojureCLR

ClojureCLR[1] – это версия Clojure для .NET CLR. Это не просто кросс-компиляция, а самостоятельный проект, цель которого обеспечить ту же степень интеграции с CLR, которая поддерживается между Clojure и JVM. Принимая во внимание различия между JVM

---

[1] https://github.com/clojure/clojure-clr; блог разработчиков этой реализации: http://clojureclr.blogspot.com.

и CLR, ClojureCLR поддерживает некоторые возможности, отсутствующие в Clojure.

Хотя ClojureCLR и не используется так же широко, как Clojure, это достаточно зрелая и хорошо протестированная реализация, представляющая собой отличный выбор для всех, кому требуется создавать приложения для CLR и пользоваться всеми удобствами Clojure.

ClojureScript

ClojureScript (https://github.com/clojure/clojurescript) – это еще одна, полностью самостоятельная реализация. Ее целевой платформой является JavaScript (точнее, ECMAScript 3), и поэтому она производит код, который может выполняться всеми современными браузерами, а также другими средами выполнения JavaScript, такими как Node.js, CouchDB[1], и другими, практически так же, как это делают CoffeeScript или Dart. Это означает, что на ClojureScript можно писать программы (используя макросы, типы, протоколы, мультиметоды и другие возможности) и развертывать их на любой платформе JavaScript.

Однако следует сразу предупредить:

1. ClojureScript – очень новая разработка. Проект был основан в 2011 году и в него до сих пор продолжают вноситься весьма существенные изменения. И все-таки, в целом это достаточно стабильная реализация; существуют проекты и сайты, использующие ClojureScript, которые действуют в нормальном эксплуатационном режиме уже много месяцев.
2. Между Clojure и ClojureScript имеется ряд нетривиальных различий. Отчасти это обусловлено значительными различиями между средами выполнения JavaScript и JVM, а отчасти потому, что ClojureScript является компилятором из исходных текстов в исходные тексты: он не создает байт-код, который мог бы быть загружен и выполнен виртуальной машиной, вместо этого компилятор ClojureScript генерирует исходный код на языке JavaScript. Обзор различий между Clojure и ClojureScript можно найти по адресу: https://github.com/clojure/clojurescript/wiki/Differences-from-Clojure.

---

[1] Из сценариев на ClojureScript можно пользоваться библиотекой Clutch, широко применявшейся в главе 15 для реализации представлений в базах данных CouchDB: https://github.com/clojure-clutch/clutch-clojurescript.

Впервые реализация ClojureScript была представлена Ричем Хикки (Rich Hickey) летом 2011, на ежемесячной встрече группы пользователей Clojure ClojureNYC[1]; видеоролик с презентацией можно найти по адресу: http://blip.tv/clojure/rich-hickey-unveils-clojurescript-5399498.

4Clojure

4Clojure (http://4clojure.com) предлагает заняться решением небольших задач программирования на Clojure, от самых простых до довольно замысловатых. Решив задачу, вы сможете посмотреть решения, предложенные другими игроками. Этот проект собрал тысячи людей, поучаствовавших в решении сотен тысяч задач.

Знакомясь с решениями, предложенными другими, имейте в виду, что подобные игры с программным кодом (code golfing: https://en.wikipedia.org/wiki/Code_golf) являются распространенной практикой на 4Clojure. Большинство трюков и идей, которые вы увидите там, являются весьма ценными; но только не считайте решения на 4Clojure авторитетным руководством по стилю программирования.

Overtone

Overtone[2] – открытая аудиосреда (audio environment), написанная на языке Clojure, которая воспроизводит звуки и музыку через сервер аудиосинтеза SuperCollider[3]. Она позволяет создавать инструменты и синтезаторы на Clojure, используя весьма высокоуровневые абстракции, и составлять из них целые композиции (добавляя, при желании, музыкальные фрагменты, которые можно найти в Интернете). Помимо возможности воспроизводить звуки и музыку программным способом, Overtone обеспечивает также поддержку TouchOSC, Monome и других устройств ввода, с помощью которых вы сможете касаниями и жестами управлять воспроизведением своего шума, созданного программным способом.

Проект Overtone был представлен одним из его лидеров на конференции Clojure Conj 2011 (http://clojure-conj.org). Мы настоятельно рекомендуем посмотреть видеоролик с презентацией, чтобы

---

[1] http://www.meetup.com/Clojure-NYC/.
[2] http://overtone.github.com.
[3] http://supercollider.sourceforge.net.

получить общее представление об истории развития этого проекта и его целях (http://blip.tv/clojure/sam-aaron-programming-music-with-overtone-5970273).

core.logic

В разделе «Включаем коллекции Clojure в работу», в главе 3, мы говорили о *реляционно-ориентированном* программировании. Библиотека `core.logic` (https://github.com/clojure/core.logic) обеспечивает поддержку разновидности логического программирования, известного как *реляционное* программирование (relational programming) и являющегося следующим шагом развития декларативного программирования. Другими примерами языков реляционного программирования являются SQL и Prolog. Библиотека `core.logic` ближе к последнему, в том смысле, что она реализует miniKanren, разновидность языка реляционного программирования, который хорошо подходит для логического программирования в ограничениях (constraint logic programming).

Благодаря дополнительной декларативности, открываются широкие возможности: *цель* (goal) в `core.logic` (реляционный аналог функции) не делает различий между параметрами и возвращаемым значением, позволяя тем самым «возвращать» их назад, поэтому вы можете получить один (или несколько) возможных аргументов, дающий известное «возвращаемое» значение[1].

Простейшим примером широты возможностей может служить `conso`, реляционный аналог `cons`. Поскольку цели («реляционные функции») не различают аргументы и возвращаемые значения, `conso` принимает свое «возвращаемое значение» в виде дополнительного третьего и последнего аргумента. Взгляните на следующий пример:

```
(use '[clojure.core.logic])

(run* [x] (conso 1 [2 3] x))      ❶
;= ((1 2 3))
(run* [x]
  (fresh [_]
```

[1] Слово «возвращаемое» взято в кавычки, чтобы подчеркнуть, что в реляционном программировании нет понятия *возвращаемое значение* в том значении, в каком оно существует в объектно-ориентированном или функциональном программировании.

```
    (conso x _ [1 2 3])))          ❷
;= (1)
(run* [x]
  (fresh [_]
    (conso _ 3 [1 2 3])))          ❸
;= ((2 3))
(run* [q]                          ❹
  (fresh [x y]
    (conso x y [1 2 3])
    (== q [x y])))
;= ((1 (2 3)))
```

❶ Сначала эта функция вызывается «очевидным» способом; возвращаемое значение – список результатов, напоминающий множество ответов, получаемых при решении уравнения, или множество результатов, получаемых при выполнении SQL-запроса. Единственным решением в этой программе является `(1 2 3)`.

❷ При вызове с двумя неизвестными аргументами (`x` и `_`), когда требуется получить потенциальное значение только для первого из них, единственным ответом будет `1`. Таким образом, `conso` может действовать как `first`.

❸ На этот раз, когда требуется вернуть только второй аргумент, единственным ответом будет `(2 3)`; то есть, `conso` может также действовать как `rest`.

❹ Можно также вернуть первые два аргумента вместе, подобно форме деструктуризации списка, такой как в `[x & [y z]]`.

Когда даже такие простые функции, как `cons` (или, скорее, `conso`) способны получить подобную выразительность и широту возможностей, это открывает новые способы изящного выражения решений сложных проблем.

Pallet

В главе 17 мы познакомились с основными способами развертывания веб-приложений на языке Clojure, но наши потребности могут оказаться намного шире. Pallet (https://github.com/pallet/pallet) – это комплект инструментов для Clojure, позволяющий решать самые разные задачи автоматизации развертывания приложений и управления вычислительной инфраструктурой. Хотя для автоматизации настройки систем все чаще применяются такие инструменты, как Chef и Puppet, цель Pallet заключается в том, чтобы предложить надмножество их функциональности, включая автоматизацию под-

готовки экземпляров для множества поставщиков услуг облачных вычислений[1]. Разумеется, так как Pallet является инструментом для Clojure, вы можете настраивать, расширять и взаимодействовать с ним (и, соответственно, с аппаратным обеспечением, серверами и процессами, образующими вычислительную инфраструктуру), используя язык Clojure и оболочку REPL, то есть, использовать все преимущества языка для настройки, системного администрирования и управления конфигурациями.

Avout

Все обсуждение конкуренции в главе 4 было сосредоточено на проблемах конкуренции внутри одного процесса, координации чередования параллельно выполняющихся процессов внутри единственной среды выполнения Clojure. Avout (http://avout.io) – это реализация атомов и ссылок Clojure (и, соответственно, транзакционной памяти для последних) для использования в распределенных приложениях. Эта реализация основана на Apache Zookeeper (http://zookeeper.apache.org), но предусматривает возможность расширения, благодаря чему допускается добавлять новые серверные (backends) реализации. Если вы находите привлекательной идею транзакционной памяти в Clojure, но вам требуется распространить семантику транзакций на несколько приложений (или экземпляров одного приложения, выполняющегося в нескольких разных процессах), Avout может оказаться естественным выбором.

Clojure на платформе Heroku

В главе 17 были представлены наиболее типичные подходы к развертыванию веб-приложений с использованием контейнеров (таких как Tomcat на Amazon Elastic Beanstalk). Однако веб-приложения на Clojure можно запускать *без* контейнеров, и, соответственно, без выполнения операций по упаковке, предполагаемых контейнерами. Хотя способы бесконтейнерного (containerless) развертывания являются относительно новыми в пространстве JVM, они получают все более широкое распространение. Одним из наиболее популяр-

---

[1] В значительной степени благодаря библиотеке jclouds, которая обеспечивает абстракцию десятков различных служб облачных вычислений: http://code.google.com/p/jclouds.

ных решений на сегодняшний день является Heroku (http://heroku.com) – платформа развертывания масштабируемых приложений, основанная на Amazon Web Services – которая в настоящее время поддерживает развертывание веб-приложений на основе библиотеки Ring с использованием Leiningen[1] и не требует выполнения отдельных этапов компиляции и упаковки.

Платформа Heroku обладает преимуществом расширения с помощью прикладных «дополнений» (add-ons) – кластеров управляемых баз данных, очередей сообщений, веб-служб, и так далее – которые можно включать и использовать внутри проекта на Clojure без необходимости настраивать и управлять ими вручную. Это – коммерческая платформа, но она прокладывает путь, показывая, как в бесконтейнерном будущем будут развертываться все приложения на Clojure.

---

[1] Подробное пошаговое руководство и документацию можно найти по адресу: http://devcenter.heroku.com/articles/clojure.

Об авторах

Чаз Эмерик (Chas Emerick) стал постоянным членом сообщества Clojure в начале 2008. Занимался разработкой ядра языка, принимал участие в десятках проектов с открытым исходным кодом на языке Clojure, а также часто выступал и писал статьи о языке Clojure и разработке ПО в целом.

Чаз занимается сопровождением проекта Clojure Atlas (http://clojureatlas.com) визуализации языка Clojure и его стандартной библиотеки с целью обучения.

Является основателем Snowtide (http://snowtide.com), небольшой компании в Западном Массачусетсе, занимающейся производством ПО. Основной круг интересов Чаза лежит в области извлечения неструктурированных данных с уклоном в обработку документов PDF. Он пишет статьи и книги о Clojure занимается разработкой ПО и предпринимательством, а также имеет другие пристрастия (http://cemerick.com).

Брайен Карпер (Brian Carper) – программист на Ruby, ставший приверженцем Clojure. Занимается программированием на Clojure, начиная с 2008, и использует этот язык и дома, и на работе для всего подряд, от разработки веб-приложений и до анализа данных в приложениях с графическим интерфейсом.

Брайен является автором приложения Gaka (https://github.com/briancarper/gaka), компилятора Clojure-to-CSS, и библиотеки объектно-реляционного отображения Oyako (https://github.com/briancarper/oyako). Пишет статьи о Clojure и на другие темы по адресу: http://briancarper.net.

Кристоф Гранд (Cristophe Grand) – давний поклонник функционального программирования, заплутавший на просторах Java, пока в начале 2008 не встретил Clojure и не влюбился в него с первого взгляда! Является автором Enlive (http://github.com/cgrand/enlive), библиотеки управления шаблонами, обеспечивающей возможность преобразования и извлечения HTML/XML; Parsley (http://github.com/cgrand/parsley), инкрементального парсер-генератора; и Mous-

tache (http://github.com/cgrand/moustache), веб-фреймворка, включающего поддержку маршрутизации и промежуточных функций для Ring.

Как независимый консультант, занимается разработкой ПО и предлагает обучение языку Clojure. Он также пишет статьи о Clojure на сайте http://clj-me.cgrand.net.

Иллюстрация на обложке

На первой странице обложки книги «Программирование на Clojure» изображен цветной бекас. Цветные бекасы (семейство Rostratulidae) делятся на три вида: большой цветной бекас, австралийский цветной бекас и южноамериканский цветной бекас.

Эти ржанки отличаются от настоящих бекасов, и, как следует из их названия, имеют более разнообразную окраску. Возможно, ближе к семействам якановых или куликов. Цветной бекас живет в болотистых местностях и у других водоемов, употребляет в пищу семена растений, рис, просо, насекомых, улиток и мелких ракообразных. Живут уединенно и «скрытно», поэтому их трудно увидеть, кроме как в брачный сезон.

Большой цветной бекас (Rostratula benghalensis) обитает в Африке, Индии и Южной Азии. Австралийский цветной бекас (Rostratula australis), долгое время считавшийся подвидом большого цветного бекаса, обитает только в Австралии и классифицируется как исчезающий вид. Эти два вида цветных бекасов проявляют половой диморфизм: самки крупнее и имеют более яркую окраску. Самцы насиживают яйца и охраняют птенцов, в то время как самки защищают свой ареал и спариваются с несколькими самцами.

Южноамериканский цветной бекас (Nycticryptes semicollaris) обитает в южной части континента. Отличается от других цветных бекасов наличием перепонок на лапках. Южноамериканские цветные бекасы образуют моногамные пары и имеют обычное распределение половых ролей, в отличие от двух других видов. В Аргентине и Чили является объектом охоты для человека.

Изображение для обложки было взято из энциклопедии «Riverside Natural History». Текст на обложке набран шрифтом Adobe ITC Garamond. Текст книги набран шрифтом Linotype Birka; для заголовков был использован шрифт Adobe Myriad Condensed, а примеры программного кода были набраны шрифтом TheSans Mono Condensed.

Предметный указатель

Символы

A

D

E

F

G

H

I

J

Q

R

S

T

U

V

W

A

Б

В

Г

Д

Е

Ж

З

И

К

Л

M

Н

О

П

P

C

T

У

Ф

Ц

Ч

Ш

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.alians-kniga.ru**.

Оптовые закупки: тел. **(499) 782-38-89**

Электронный адрес: **books@alians-kniga.ru**.

Чаз Эмерик, Брайен Карпер, Кристоф Гранд

Программирование на Clojure

Практика применения Lisp в мире Java

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 51. Тираж 100 экз.

Веб-сайт издательства: www.дмк.рф