Содержание

Предисловие к первому изданию

Реляционные базы данных, бесспорно, составляют основу современного предприятия. В то время как современные языки программирования, включая Java, обеспечивают интуитивное, объектно-ориентированное представление бизнес-сущностей уровня приложения, данные, лежащие в основе этих сущностей, имеют выраженную реляционную природу. Кроме того, главное преимущество реляционной модели перед более ранней навигационной моделью, а также поздними моделями объектно-ориентированных баз данных – в том, что она изначально внутренне независима от программных взаимодействий и представления данных на уровне приложения. Было предпринято немало попыток для объединения реляционной и объектно-ориентированной технологий или замещения одной на другую, но пропасть между ними остается сегодня одним из непреложных фактов. Именно эту задачу – обеспечить связь между реляционными данными и Java-объектами – решает Hibernate при помощи своего подхода к реализации объектно-реляционного отображения (Object/Relational Mapping, ORM). Hibernate решает данную задачу очень прагматичным, ясным и реалистичным способом.

Как показывают Кристиан Бауэр (Christian Bauer) и Гэвин Кинг (Gavin King) в этой книге, эффективное использование технологии ORM в любом бизнес-окружении, кроме простейшего, требует понимания особенностей работы механизма, осуществляющего посредничество между реляционными данными и объектами. То есть разработчик должен понимать требования к своему приложению и к данным, владеть языком SQL, знать структуры реляционных хранилищ, а также иметь представление о возможных способах оптимизации, предоставляемых реляционными технологиями. Hibernate не только предоставляет полностью рабочее решение, непосредственно удовлетворяющее этим требованиям, но и гибкую и настраиваемую архитектуру. Разработчики Hibernate изначально сделали фреймворк модульным, расширяемым и легко настраиваемым под нужды пользователя. В результате через несколько лет после выхода первой версии фреймворк Hibernate быстро стал – и заслуженно – одной из ведущих реализаций ORM-технологий для разработчиков корпоративных приложений.

В этой книге представлен подробный обзор фреймворка Hibernate. Описывается, как использовать его возможности отображения типов и средства моделирования ассоциаций и наследования; как эффективно извлекать объекты, используя

язык запросов Hibernate; как настраивать Hibernate для работы в управляемом и неуправляемом окружениях; как использовать его инструментарий. Кроме того, на протяжении всей книги авторы приоткрывают проблемы ORM и проектные решения, легшие в основу Hibernate. Это дает читателю глубокое понимание эффективного использования ORM как корпоративной технологии. Данная книга является подробным руководством по Hibernate и объектно-реляционному отображению в корпоративных вычислениях.

Линда Демишель (*Linda DeMichiel*)
Ведущий архитектор, Enterprise Javabeans
Sun Microsystems
Ноябрь 2012

Введение

Это наша третья книга о Hibernate, проекте с открытым исходным кодом, которому почти 15 лет. Согласно недавнему опросу, Hibernate оказался в числе пяти самых популярных инструментов, которыми Java-разработчики пользуются каждый день. Это говорит о том, что базы данных SQL являются предпочтительной технологией для надежного хранения и управления данными, особенно в области разработки корпоративных приложений на Java. Также это является доказательством качества доступных спецификаций и инструментов, упрощающих запуск проектов, оценку и снижение рисков при создании крупных и сложных приложений.

Сейчас доступны пятая версия Hibernate и вторая версия спецификации Java Persistence API (JPA), которую реализует Hibernate. Ядро Hibernate или то, что сейчас называется объектно-реляционным отображением (Object/Relational Mapping, ORM), уже долгое время является развитой технологией, и на протяжении многих лет было сделано множество мелких улучшений. Другие связанные проекты, такие как Hibernate Search, Hibernate Bean Validation и недавнее объектно-сеточное отображение (Object/Grid Mapping, OGM), привносят новые инновационные решения, которые превращают Hibernate в полноценный набор инструментов для решения широкого спектра задач управления данными.

Когда мы работали над предыдущим изданием этой книги, с Hibernate происходили важные изменения: в силу своего органичного развития, влияния со стороны сообщества разработчиков свободного ПО и повседневных требований Java-разработчиков Hibernate пришлось стать более формальным и реализовать первую версию спецификации JPA (Java Persistence API). Поэтому предыдущее издание получилось громоздким, так как многие примеры мы были вынуждены демонстрировать с использованием старого и нового, стандартизированного, подходов.

В настоящий момент этот расхождение практически исчезло, и мы можем в первую очередь опираться на стандартизированный прикладной программный интерфейс (API) и архитектуру Java Persistence. Также в этом издании мы обсудим множество выдающихся особенностей Hibernate. Хотя объем книги уменьшился, по сравнению с предыдущим изданием, мы использовали это место для многочисленных новых примеров. Мы также рассмотрим, как JPA вписывается в общую картину Java EE и как ваше приложение может интегрировать Bean Validation, EJB, CDI и JSF.

Пусть это новое издание станет путеводителем для вашего первого проекта Hibernate. Мы надеемся, что оно заменит предыдущее издание в качестве настольного справочного материала по Hibernate.

Об этой книге

Эта книга является и руководством, и справочником по Hibernate и Java Persistence. Если вы новичок в Hibernate, рекомендуем читать, начиная с главы 1, и опробовать все примеры с «Hello World» в главе 2. Если вы использовали старую версию Hibernate, прочитайте бегло две первые главы, чтобы получить общее представление, и переходите к середине главы 3. Там, где необходимо, мы сообщим, если конкретный раздел или тема являются дополнительными или справочными, которые можно пропустить при первом чтении.

Структура книги

Эта книга состоит из пяти больших частей.

В части I «Начинаем работать с ORM» мы обсудим основы объектно-реляционного отображения. Пройдемся по практическому руководству, чтобы помочь вам создать первый проект с Hibernate. Рассмотрим проектирование Java-приложений с точки зрения создания моделей предметной области и познакомимся с вариантами определения метаданных для объектно-реляционного отображения.

В части II «Стратегии отображения» рассказывается о классах Java и их свойствах, а также об их отображении в таблицы и столбцы SQL. Мы рассмотрим все основные и продвинутые способы отображения в Hibernate и Java Persistence. Покажем, как работать с наследованием, коллекциями и сложными ассоциациями классов. В заключение обсудим интеграцию с существующими схемами баз данных, а также особенно запутанные стратегии отображения.

Часть III «Транзакционная обработка данных» посвящена загрузке и сохранению данных с помощью Hibernate и Java Persistence. Мы представим программные интерфейсы, способы создания транзакционных приложений и то, как эффективнее загружать данные из базы данных в Hibernate.

В части IV «Создание запросов» мы познакомимся с механикой извлечения данных и детально рассмотрим язык запросов и прикладной программный интерфейс (API). Не все главы данного раздела написаны как руководство; мы рассчитываем, что вы будете часто заглядывать в эту часть книги во время разработки приложений для поиска решений проблем с конкретными запросами.

В части V «Разработка приложений» мы обсудим проектирование и разработку многоуровневых приложений баз данных на Java. Обсудим наиболее распространенные шаблоны проектирования, используемые вместе с Hibernate, такие как «Объект доступа к данным» (Data Access Object, DAO). Вы увидите, как можно легко протестировать приложение, использующее Hibernate, и познакомитесь с другими распространенными приемами, используемыми при работе с инструментами объектно-реляционного отображения в веб-приложениях или клиент-серверных приложениях в целом.

Кому адресована эта книга?

Читатели данной книги должны быть знакомы с основами объектно-ориентированного программирования и иметь практические навыки его применения. Чтобы понять примеры приложений, вы должны быть знакомы с языком программирования Java и унифицированным языком моделирования (Unified Modeling Language, UML).

В первую очередь книга адресована Java-программистам, работающим с базами данных SQL. Мы покажем, как повысить продуктивность при работе с ORM. Если вы – разработчик баз данных, эта книга может отчасти послужить вам введением в объектно-ориентированную разработку программного обеспечения.

Если вы администратор баз данных (DBA), вас наверняка заинтересует влияние ORM на производительность, а также способы настройки СУБД SQL и уровня хранения данных для достижения желаемой производительности. Доступ к данным является узким местом любого Java-приложения, поэтому проблемам производительности в данной книге уделяется повышенное внимание. Многие DBA, по понятным причинам, испытывают тревогу, не веря в высокую производительность автоматически сгенерированного кода SQL; мы постараемся развеять эти страхи, а также указать случаи, когда в приложениях не следует использовать автоматического доступа к данным. Вы почувствуете себя свободнее, когда поймете, что мы не считаем ORM лучшим решением для каждой проблемы.

Соглашения об оформлении программного кода

Эта книга изобилует примерами, включающими всевозможные артефакты Hibernate-приложений: Java-код, файлы конфигурации Hibernate и XML-файлы с метаданными отображений. Исходный код в листингах или тексте `выделен моноширинным шрифтом, как этот`, чтобы отделить его от остального текста. Кроме того, имена Java-методов, параметры компонентов, свойства объектов, а также XML-элементы и их атрибуты в тексте представлены с использованием `моноширинного шрифта`.

Листинги на Java, XML и HTML могут быть объемными. Во многих случаях исходный код (доступный онлайн) был переформатирован; мы добавили разделители строк и переделали отступы, чтобы уместить их по ширине книжных страниц. В редких случаях, когда этого оказалось недостаточно, в листинги были добавлены символы продолжения строки (➥). Также из многих листингов, описываемых в тексте, мы убрали комментарии. Некоторые листинги сопровождают аннотации, выделяющие важные понятия. В других случаях используются пронумерованные маркеры, отсылающие к пояснениям в тексте, следующим за листингами.

Загрузка исходного кода

Hibernate – это проект с открытым исходным кодом, распространяемым на условиях лицензии Lesser GNU Public Lisence. Инструкции по загрузке модулей Hibernate в виде исходных или двоичных кодов доступны на сайте проекта: http://

hibernate.org/. Исходный код всех примеров в данной книге доступен на http://jpwh.org. Код примеров можно также загрузить с сайта издательства https://www.manning.com/books/java-persistence-with-hibernate-second-edition.

Автор онлайн

Приобретая книгу «Java Persistence API и Hibernate», вы получаете бесплатный доступ на частный веб-форум издательства Manning Publications, где вы сможете оставлять отзывы о книге, задавать технические вопросы и получать помощь от авторов и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в браузере страницу https://www.manning.com/books/java-persistence-with-hibernate-second-edition. На этой странице «Author Online» (Автор в сети) описывается, как попасть на форум после регистрации, какие виды помощи доступны и правила поведения на форуме.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны автора отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – его присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать автору стимулирующие вопросы, чтобы его интерес не угасал!

Форум и архивы предыдущих дискуссий будут оставаться доступными, пока книга продолжает издаваться.

Об авторах

Кристиан Бауэр (Christian Bauer) – член коллектива разработчиков Hibernate; он работает инструктором и консультантом.

Гэвин Кинг (Gavin King) – основатель проекта Hibernate и член первоначального состава экспертной группы по Java Persistence (JSR 220). Он также участвовал в работе по стандартизации CDI (JSR 299). В настоящее время Гэвин разрабатывает новый язык программирования Ceylon.

Гэри Грегори (Gary Gregory) является главным инженером в Rocket Software, где работает над серверами приложений и интеграцией с устаревшими системами. Еще он соавтор книг издательства Manning: «JUnit in Action» и «Spring Batch in Action», а также член комитетов по руководству проектом (Project Management Commitee) в различных проектах в Apache Software Foundation: Commons, HttpComponents, Logging Services и Xalan.

Об изображении на обложке

Иллюстрация на обложке книги «Java Persistence API и Hibernate, второе издание» взята из сборника костюмов Османской империи, изданного 1 января 1802 г. Вильямом Миллером (William Miller) в Old Bond Street, Лондон. Титульный лист сборника утерян, поэтому мы не смогли точно определить дату его выхода. В содержании книги изображения описаны на английском и французском языках, и под каждой иллюстрацией приводятся имена двух художников, которые трудились над ней. Не сомневаемся, что все они, несомненно, были бы удивлены, узнав, что их творчество украсит обложку книги по программированию... 200 лет спустя.

Рисунки из сборника костюмов Османской империи, так же как и другие иллюстрации, появляющиеся на наших обложках, возрождают богатство и разнообразие традиций в одежде двухсотлетней давности. Они напоминают о чувствах уединенности и удаленности этого периода – и любого другого исторического периода, кроме нашего гиперкинетического настоящего. Манеры одеваться сильно изменились с тех пор, а своеобразие каждого региона, такое яркое в то время, давно поблекло. Сейчас бывает трудно различить обитателей разных континентов. Возможно, если смотреть на это оптимистично, мы обменяли культурное и визуальное разнообразие на более разнообразную частную жизнь или более разнообразную интеллектуальную и техническую жизнь.

Мы в Manning высоко ценим изобретательность, инициативу и, конечно, радость от компьютерного бизнеса с книжными обложками, основанными на разнообразии жизни в разных регионах два века назад, которое оживает благодаря картинкам из этой коллекции.

Часть I

НАЧИНАЕМ РАБОТАТЬ С ORM

В части I мы объясним, почему долговременное хранение объектов является такой сложной темой, и расскажем о некоторых практических решениях. В главе 1 описывается несоответствие объектной и реляционной парадигм и приводится несколько стратегий для преодоления этого несоответствия – в первую очередь объектно-реляционное отображение (ORM). В главе 2 мы по шагам разберем с вами учебный пример с Hibernate и Java Persistence – программу «Hello World». После такой начальной подготовки в главе 3 вы узнаете, как проектировать и разрабатывать сложные предметные модели на Java и какие виды метаданных отображения доступны.

После прочтения этой части книги вы поймете, для чего требуется ORM и как Hibernate и Java Persistence работают на практике. Вы создадите свой первый небольшой проект и подготовитесь к решению более сложных задач. Вы также узнаете, как представлять реальные бизнес-сущности в виде предметных моделей на Java, и выберете наиболее предпочтительный формат для работы с метаданными ORM.

Глава 1

Основы объектно-реляционного отображения

В этой главе:

- использование баз данных SQL в Java-приложениях;
- несоответствие объектной и реляционной парадигм;
- знакомство с ORM, JPA и Hibernate.

Эта книга рассказывает о Hibernate, поэтому все свое внимание мы сосредоточим на Hibernate – одной из реализаций спецификации Java Persistence API. Мы рассмотрим основные и продвинутые особенности и опишем несколько способов создания Java-приложений с использованием Java Persistence. Часто эти рекомендации будут относиться не только к Hibernate. Иногда это будут наши идеи о *наилучших* способах работы с хранимыми данными, проиллюстрированные в контексте Hibernate.

Во всех программных проектах, над которыми мы работали, подход к управлению хранимыми данными являлся ключевым проектным решением. Учитывая, что долговременное хранение данных не является необычным требованием к Java-приложениям, можно предположить, что выбор между схожими и проверенными решениями довольно прост. Возьмите, к примеру, фреймворки веб-приложений (JavaServer Faces, Struts или GWT), библиотеки компонентов пользовательского интерфейса (Swing или SWT) или процессоры шаблонов (JSP или Thymeleaf). Каждое из решений имеет свои достоинства и недостатки, но у них одна область применения и общий подход. К сожалению, с технологиями долговременного хранения данных дела обстоят иначе, где схожие проблемы часто решаются совершенно разными способами.

Долговременное хранение всегда было темой горячего обсуждения в Java-сообществе. Является ли хранение данных проблемой, уже решенной с помощью SQL и хранимых процедур, или же она является более широкой задачей, решать кото-

рую нужно, используя специальные компонентные модели Java, такие как EJB? Следует ли вручную кодировать даже самые примитивные CRUD-операции (создание, чтение, изменение, удаление) или они должны быть автоматизированы? Как обеспечить переносимость, если в каждой СУБД используется свой диалект SQL? Следует ли полностью отказаться от SQL и принять иные технологии баз данных, такие как системы управления объектными базами данных или NoSQL? Этот спор может никогда не утихнуть, но решение под названием *объектно-реляционное отображение* (ORM) уже получило широкое признание во многом благодаря инновациям Hibernate – реализации ORM с открытым исходным кодом.

Прежде чем начать работать с Hibernate, следует понять коренную проблему долговременного хранения объектов и ORM. В этой главе объясняется, зачем нужны такие инструменты, как Hibernate, и такие спецификации, как Java Persistence API (JPA).

Сперва мы определим, что подразумевается под управлением хранимыми данными в контексте объектно-ориентированных приложений, и обсудим взаимосвязь SQL, JDBC и Java – технологий и стандартов, лежащих в основе Hibernate. Затем обсудим так называемое *несоответствие объектной и реляционной парадигм* и базовую проблему, возникающую в объектно-ориентированной разработке программного обеспечения с использованием баз данных SQL. Эти проблемы ясно указывают, что нам нужны инструменты и шаблоны, уменьшающие время разработки кода, связанного с долговременным хранением данных, в наших приложениях.

Лучший способ изучения Hibernate не обязательно должен быть последовательным. Мы понимаем, что вам, возможно, захочется попробовать Hibernate сразу же. Если вы намерены поступить именно так, переходите к следующей главе и разверните проект примера «Hello World». Мы советуем вам после этого вернуться к данной главе, чтобы овладеть всеми базовыми понятиями, необходимыми для освоения остального материала.

1.1. Что такое долговременное хранение?

Почти все приложения используют данные, хранящиеся долговременно (persistent data). Долговременное хранение является одним из фундаментальных понятий в разработке приложений. Если бы информационная система не сохраняла данных во время отключения, от нее было бы мало толку. *Долговременное хранение объектов* (object persistence) означает, что отдельные объекты могут существовать дольше, чем процесс приложения; они могут помещаться в хранилище данных и восстанавливаться впоследствии. Когда мы говорим о долговременном хранении в Java, мы обычно имеем в виду отображение и сохранение отдельных объектов в базе данных SQL. Мы начнем с краткого обзора технологии и ее применения в Java. Вооруженные этими знаниями, мы продолжим обсуждение технологии долговременного хранения и ее реализации в объектно-ориентированных приложениях.

1.1.1. Реляционные базы данных

Возможно, вы, как и большинство разработчиков программного обеспечения, работали с SQL и реляционными базами данных; многие из нас сталкиваются с подобными системами ежедневно. Системы управления реляционными базами данных имеют прикладной программный интерфейс, основанный на SQL, поэтому мы называем современные продукты, относящиеся к реляционным базам данных, *системами управления базами данных* SQL (СУБД), или, говоря о конкретной системе, *базами данных* SQL.

Реляционные технологии хорошо известны, и один этот факт является достаточным аргументом в их пользу для многих организаций. Но упомянуть лишь об этом означало бы выразить меньшее почтение, чем следует. Сила реляционных баз данных – в невероятно гибком и устойчивом подходе к управлению данными. Благодаря тщательно исследованным теоретическим обоснованиям реляционной модели, помимо других желаемых качеств, реляционные базы данных способны обеспечивать и защищать целостность хранимых данных. Вам, возможно, известна работа Э. Кодда (E. F. Codd), вышедшая четыре десятилетия назад, – введение в реляционную модель «A Relational Model of Data for Large Shared Data Banks» (Codd, 1970). Более поздней работой, которую можно порекомендовать для прочтения, посвященной SQL, является труд К. Дейта (C. J. Date) «SQL and Relational Theory»[1] (Date, 2009).

Реляционные СУБД могут использоваться не только с Java, так же как и базы данных SQL могут использоваться не только конкретным приложением. Этот важный принцип носит название *независимости данных* (data independence). Другими словами, мы бы хотели как можно ярче подчеркнуть этот факт: *данные существуют дольше, чем любое приложение*. Реляционные технологии дают возможность совместного использования данных разными приложениями или разными частями одной общей системы (например, приложением ввода данных и приложением отчетов). Реляционные технологии являются общим знаменателем для различных систем и технологических платформ. В результате реляционная модель обычно становится общим основанием для представления бизнес-сущностей в масштабах предприятия.

Прежде чем переходить к подробному обсуждению практических сторон баз данных SQL, следует отметить важный момент: система управления базами данных, которая позиционируется на рынке как реляционная, но которая предоставляет только интерфейс SQL, не является в действительности реляционной и во многом даже близко не соответствует изначальной идее. Естественно, это привело к путанице. Профессиональные разработчики SQL критикуют реляционную модель за ограничения в языке SQL, а эксперты в управлении реляционными данными критикуют стандарт SQL как слабое воплощение реляционной модели и идеалов. Разработчики приложений находятся где-то посередине, решая труд-

---

[1] *Дейт К. Дж.* SQL и реляционная теория. Как грамотно писать код на SQL. ISBN: 978-5-93286-173-8. Символ-Плюс, 2010. – *Прим. ред.*

ную задачу по выпуску чего-то работоспособного. Далее в книге мы будем подчеркивать важные и значимые стороны этой проблемы, но в целом мы сосредоточимся на решении практических задач. Если вам хочется узнать больше, мы очень рекомендуем «Practical Issues in Database Management: A Reference for the Thinking Practicioner» (Pascal, 2000) Фабиана Паскаля (Fabian Pascal) и «Introduction to Database Systems» (Date, 2003) Кристофера Дейта (Chris Date). В этих книгах вы ознакомитесь с теорией, понятиями и идеалами (реляционных) систем управления базами данных. Последняя книга является прекрасным справочником (очень объемным) по всем вопросам, касающимся баз данных и управления данными.

1.1.2. Разбираемся с SQL

Для эффективного использования Hibernate необходимо глубокое понимание реляционной модели и SQL. Вы должны понимать реляционную модель и такие темы, как нормализация для обеспечения целостности данных, а также использовать знание SQL для оптимизации производительности приложения с Hibernate. Hibernate автоматизирует решение множества повторяющихся задач, но ваше знание технологии долговременного хранения должно распространяться дальше Hibernate, если хотите воспользоваться всеми преимуществами современных баз данных SQL. Ознакомьтесь со списком литературы в конце книги для более глубокого погружения в тему.

Возможно, вы использовали SQL в течение многих лет и знакомы с основными операциями и инструкциями. Тем не менее, опираясь на собственный опыт, мы можем утверждать, что код SQL иногда трудно запомнить, а некоторые термины имеют разное значение.

Рассмотрим некоторые термины SQL, используемые в книге. Язык SQL широко используется как *язык описания данных* (Data Definition Language, DDL) при *создании* (create), *изменении* (alter) или *удалении* (drop) таких артефактов, как таблицы и ограничения в каталоге СУБД. При наличии готовой *схемы* SQL используется как *язык управления данными* (Data Manipulation Language, DML) для *вставки* (insertion), *изменения* (update) и *удаления* (delete) этих данных. Для извлечения данных используются запросы с *ограничениями* (restrictions), *проекциями* (projections) и *декартовыми произведениями* (Cartesian products). Для лучшего представления с помощью SQL данные можно *соединять* (join), *агрегировать* (aggregate) и *группировать* (group) по необходимости. Можно даже помещать одни инструкции SQL внутрь других – эта техника носит название *вложенных запросов* (subselects). Когда меняются бизнес-требования, приходится менять схему базы данных при помощи инструкций DDL уже после того, как были сохранены данные; это называется *эволюцией схемы* (schema evolution).

Тем, кто давно использует SQL и желает узнать больше об оптимизации и о том, как выполняются инструкции SQL, можно порекомендовать отличную книгу «SQL Tuning» (Tow, 2003) Дэна Тоу (Dan Tow). Книга «SQL Antipatterns: Avoiding the Pitfalls of Database Programming» (Karwin, 2010) является хорошим ресурсом для изучения практического применения SQL на примерах некорректного использования SQL.

Несмотря на то что база данных SQL является одной из частей ORM, другая ее часть состоит из данных Java-приложения, которые необходимо хранить и извлекать из базы данных.

1.1.3. Использование SQL в Java

При работе с базой данных SQL в Java-приложении инструкции SQL передаются в базу данных через прикладной интерфейс Java Database Connectivity (JDBC). Независимо от того, как написан код SQL – вручную, включен в Java-код или создан Java-кодом «на лету», для привязки аргументов к параметрам запроса, выполнения запроса, итераций по результатам запроса, извлечения значений из результирующей выборки и т. д. будет использован JDBC API. Все это – низкоуровневые задачи доступа к данным; как разработчики приложений мы больше заинтересованы в решении предметной задачи, требующей доступа к данным. Что бы мы действительно хотели писать – так это код, сохраняющий и возвращающий экземпляры классов, который избавил бы нас от этой низкоуровневой рутины.

Из-за того, что задачи доступа к данным обычно такие нудные, мы спрашиваем – действительно ли реляционная модель данных и (особенно) SQL являются правильным решением для долговременного хранения данных в объектно-ориентированных приложениях? Мы отвечаем на этот вопрос однозначно – да! Есть много причин, почему базы данных SQL доминируют в индустрии ПО, – реляционные системы управления базами данных являются единственной испытанной универсальной технологией, и они почти всегда *требуются* в Java-проектах.

Обратите внимание, что мы не утверждаем, будто реляционная технология *всегда* является лучшим решением. Существует множество требований к управлению данными, вынуждающих использовать иные подходы. Например, распределенные системы в масштабах Интернета (поисковые системы, сети распространения контента, пиринговые сети обмена информацией, обмен мгновенными сообщениями) сталкиваются с исключительно большим количеством транзакций. В большинстве из них не требуется, чтобы после изменения данных все процессы видели одно и то же обновленное состояние (сильная транзакционная согласованность). Пользователям может быть достаточно слабой согласованности, когда после изменения может возникнуть окно несогласованности, прежде чем все процессы получат обновленные данные. Многие научные приложения работают с огромными, но специализированными наборами данных. Подобные системы с их уникальными проблемами требуют таких же уникальных и, как правило, нестандартных решений проблемы хранения данных. Универсальные инструменты управления данными, такие как базы данных SQL (поддерживающие транзакции, которые соответствуют требованиям ACID[1]), JDBC и Hibernate, здесь играют второстепенную роль.

---

[1] https://ru.wikipedia.org/wiki/ACID/. – *Прим. ред.*

Реляционные системы в масштабах Интернета

Чтобы понять, почему реляционные системы и связанные с ними гарантии целостности плохо масштабируются, мы рекомендуем прежде познакомиться с теоремой CAP, согласно которой распределенная система не может быть *согласованной*, *доступной* и *устойчивой к отказам разделов* одновременно[1].

Система может гарантировать актуальность данных на все узлах одновременно и надежную обработку всех запросов чтения и записи. Но когда часть системы может оказаться недоступной из-за проблем с узлом, сетью или дата-центром, следует отказаться от сильной согласованности (линеаризуемости) или 100%-ной доступности. На практике это означает, что необходима стратегия, которая бы отслеживала отказы разделов и до определенного уровня восстанавливала либо согласованность, либо доступность (например, сделав часть системы недоступной для синхронизации в фоновом режиме). Обычно необходимость в сильной согласованности зависит от данных, пользователя или операции.

Примерами легко масштабируемых реляционных СУБД могут служить VoltDB (https://www.voltdb.com/) и NuoDB (https://www.nuodb.com/). Кроме того, в весьма интересной статье «F1 – The-Fault-TolerantDistributedRDBMSSupportingGoogleAd's Business» (Shute, 2012) вы узнаете, как Google масштабирует свою самую главную базу данных для рекламного бизнеса и почему она является реляционной/SQL.

В этой книге мы будем рассматривать проблемы хранения данных и их совместного использования в контексте объектно-ориентированных приложений, основанных на *модели предметной области* (domain model). Вместо работы непосредственно со строками и колонками в экземплярах `java.sql.ResultSet` бизнес-логика приложения взаимодействует с объектно-ориентированной моделью предметной области конкретного приложения. Если в схеме базы данных SQL для онлайн-аукциона имеются таблицы `ITEM` (лот) и `BID` (предложение цены), в Java-приложении могут быть определены классы `Item`и`Bid`. Вместо того чтобы читать и записывать значения конкретных строк и колонок с помощью класса `ResultSet`, приложение загружает и сохраняет экземпляры классов `Item` и `Bid`.

То есть в процессе выполнения приложение взаимодействует с экземплярами данных классов. Каждый экземпляр `Bid` ссылается на экземпляр `Item` аукциона, а каждый экземпляр `Item` может иметь множество ссылок на экземпляры `Bid`. Бизнес-логика выполняется не на стороне базы данных (в виде хранимой процедуры); она реализована на Java и выполняется на уровне приложения. Это позволяет ей использовать такие сложные понятия ООП, как наследования и полиморфизм. К примеру, мы могли бы использовать такие известные шаблоны проектирования, как «Стратегия» (Strategy), «Посредник» (Mediator) и «Компоновщик» (Composite), описание которых можно найти в «Design Patterns: Elementsof Reusable Object-Oriented Software» [Gamma, 1995][2], опирающиеся на полиморфные вызовы методов.

---

[1] https://ru.wikipedia.org/wiki/Теорема_CAP. – *Прим. ред.*

[2] *Гамма Э.* Приемы объектно-ориентированного проектирования. ISBN: 978-5-496-00389-6. Питер, 2013. – *Прим. ред.*

Но есть нюанс: не все Java-приложения спроектированы подобным образом, да и не всегда это оправдано. Простые приложения прекрасно обойдутся без предметной модели. Используйте JDBC `ResultSet`, если это все, что вам нужно. Вызывайте хранимые процедуры и читайте возвращаемые ими наборы данных. Многим приложениям требуется выполнять процедуры, модифицирующие большие объемы данных. Вы можете реализовать отчеты, используя обычные запросы SQL, и выдавать результат прямо на экран. SQL и JDBC API отлично подходят для работы с данными в табличном представлении, а JDBC `RowSet` сильно упрощает операции CRUD. Работа с таким представлением хранимых данных довольно проста и понятна.

Но в случае, когда приложение имеет нетривиальную бизнес-логику, использование модели предметной области поможет сильно улучшить повторное использование кода и простоту сопровождения.

На протяжении десятилетий разработчики говорят о *несоответствии парадигм*. Это несоответствие объясняет, почему многие корпоративные проекты тратят так много усилий на проблемы, касающиеся хранения информации. *Парадигмы*, о которых идет речь, – это объектное и реляционное моделирование, а на практике – ООП и SQL.

Осознав это, вы начнете видеть проблемы – некоторые хорошо изученные, а некоторые не очень, – которые приложение должно решать, объединяя оба подхода: объектно-ориентированное моделирование предметной области и реляционное моделирование хранимых данных. Давайте взглянем поближе на это так называемое несоответствие парадигм.

1.2. Несоответствие парадигм

Несоответствие объектной и реляционной парадигм можно разделить на несколько частей, каждую из которых мы рассмотрим отдельно. Начнем с примера, который не имеет проблемы. По мере того как мы будем его развивать, вы увидите, как начнет проявляться несоответствие.

Предположим, вам необходимо разработать приложение для электронной коммерции. Приложению нужен класс, представляющий информацию о пользователе системы, и еще один класс, представляющий информацию о платежных реквизитах пользователя, как показано на рис. 1.1.

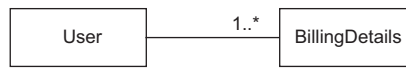

Рис. 1.1 ❖ Простая UML-диаграмма сущностей `User` и `Billing Details`

Как видно на диаграмме, у пользователя (класс `User`) может быть несколько платежных реквизитов (класс `BillingDetails`). Вы можете осуществлять навигацию по связи между классами в обоих направлениях; это означает, что можно

выполнять итерации по коллекциям или вызывать методы для получения «другой» стороны отношения. Классы, выражающие это отношение, могут быть очень простыми:

```
public class User {

    String username;
    String address;
    Set billingDetails;

// Accessor methods (getter/setter), business methods, etc.
}

public class BillingDetails {

String account;
String bankname;
User user;

// Accessor methods (getter/setter), business methods, etc.
}
```

Отметим, что нас интересуют только состояния сущностей, поэтому мы опустили реализацию методов доступа и бизнес-методов, таких как `getUsername()` или `billAuction()`.

В данном случае не составит труда придумать схему SQL:

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS varchar(255) not null
);

create table BILLINGDETAILS (
    ACCOUNT varchar(15) not null primary key,
    BANKNAME varchar(255) not null,
    USERNAME varchar(15) not null,
    foreign key (USERNAME) references USERS
);
```

Столбец `USERNAME` в таблице `BILLINGDETAILS`, являющийся внешним ключом, представляет отношение между двумя сущностями. В такой простой предметной модели трудно различить несоответствие объектной и реляционной парадигм; мы легко сможем написать JDBC-код для вставки, изменения и удаления информации о пользователях и платежных реквизитах.

Теперь рассмотрим более реалистичный пример. Несоответствие парадигм проявится после того, как в приложении будет больше сущностей и их отношений.

1.2.1. Проблема детализации

Наиболее очевидная проблема текущей реализации в том, что адрес представлен простым значением типа `String`. В большинстве систем необходимо раздельно хранить улицу, город, штат, страну и почтовый индекс. Конечно, все эти свойства

можно добавить прямо в класс User, но, поскольку велика вероятность, что и другие классы системы будут использовать информацию об адресах, имеет смысл создать класс Address. На рис. 1.2 показана обновленная модель:

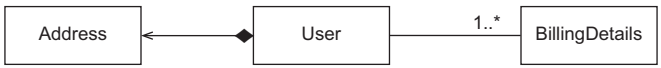

Рис. 1.2 ❖ У пользователя (User) есть адрес (Address)

Следует ли создать таблицу ADDRESS? Не обязательно; обычно информация об адресе хранится в таблице USERS, но в отдельных столбцах. Такое решение более производительно, так как не требует выполнять соединения таблиц, чтобы получить пользователя и адрес одним запросом. Самым элегантным решением были бы создание нового типа данных SQL, представляющего адрес, и добавление одного столбца этого типа в таблицу USERS вместо нескольких отдельных колонок.

Теперь имеется выбор между добавлением нескольких столбцов или одного (нового типа данных SQL). Это, очевидно, является проблемой *детализации* (problem of granularity). В широком смысле детализация относится к размерам типов, с которыми вы работаете.

Вернемся к нашему примеру. Добавление нового типа данных в каталог базы данных для хранения экземпляров Java-класса Address в одном столбце выглядит лучшим решением:

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS address not null
);
```

Новый тип (класс) Addressв Java и ADDRESS, новый тип данных SQL, должны обеспечить взаимодействие. Но вы обнаружите множество проблем, если проверите поддержку типов, определяемых пользователем (User-defined Data Types, UDT), в современных системах управления базами данных SQL.

Поддержка UDT – это одно из так называемых *объектно-реляционных расширений* традиционного SQL. Этот термин сбивает с толку, так как означает, что СУБД имеет (или должна поддерживать) сложную систему типов – то, что вы принимаете за данность, если кто-то продает вам систему, способную управлять данными в реляционном стиле. К сожалению, поддержка UDT является малопонятной особенностью многих СУБД SQL и, определенно, не является переносимой между различными продуктами. Более того, стандарт SQL слабо поддерживает типы, определяемые пользователем.

Это ограничение не является недостатком реляционной модели данных. Провал в стандартизации такой важной функциональности вы можете рассматривать как последствие войн между производителями объектно-реляционных баз данных в середине 90-х. Сегодня большинство инженеров искренне считает, что SQL-системы обладают ограниченной системой типов. Даже при наличии слож-

ной системы UDT в вашей СУБД вам наверняка придется объявлять новые типы в Java, а затем дублировать их в SQL. Попытки поиска лучшей альтернативы в среде Java, как, например, SQLJ, к сожалению, не увенчались успехом. СУБД редко поддерживают развертывание и выполнение Java-классов непосредственно в базе данных, но даже если такая поддержка доступна, она обычно ограничена базовой функциональностью и сложна для повседневного применения.

По этой и многим другим причинам использование UDT или типов Java в базах данных SQL еще не стало общепринятой практикой в отрасли; маловероятно, что вы столкнетесь со старой схемой, в которой широко применяются UDT. Следовательно, вы не сможете, и не будете, хранить экземпляры нового класса `Address` в одном столбце, имеющем тот же тип данных, что и в Java.

Более практичное решение этой проблемы: создать несколько столбцов предопределенных типов (логического, числового, строкового), встроенных производителем. Согласно ему, таблицу USERS можно определить следующим образом:

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS_STREET varchar(255) not null,
    ADDRESS_ZIPCODE varchar(5) not null,
    ADDRESS_CITY varchar(255) not null
);
```

Классы предметной модели в Java имеют различную степень детализации: от более крупных классов сущностей, как `User`, до более детализированных классов, как `Address`, и простого `SwissZipCode`, расширяющего `AbstractNumericZipCode` (в зависимости от требуемого уровня абстракции). В базе данных SQL, напротив, доступны лишь два уровня детализации: реляционные типы, созданные вами, как, например, `USERS` и `BILLINGDETAILS`, и встроенные, такие как `VARCHAR`, `BIGINT` или `TIMESTAMP`.

Большинство механизмов хранения не замечает этого несоответствия и в итоге навязывает менее гибкое представление SQL-систем объектно-ориентированным системам, делая их более плоскими.

Оказывается, что проблему детализации легко решить. Мы бы даже не стали ее обсуждать, если бы не тот факт, что она проявляется во многих существующих системах. Мы опишем решение этой проблемы в разделе 4.1.

Более сложная и интересная проблема возникает, когда мы имеем дело с моделями предметной области, основанными на *наследовании* – принципе объектно-ориентированного проектирования, который вы могли бы использовать, чтобы выставлять счета пользователям вашего приложения для электронной коммерции новыми и интересными способами.

1.2.2. Проблема подтипов

Наследование типов в Java реализуется при помощи суперклассов и подклассов. Чтобы продемонстрировать, почему это может представлять проблему несоответствия, давайте расширим возможности приложения, чтобы можно было произ-

водить оплату не только с банковского счета, но и при помощи кредитных или дебетовых карт. Наиболее естественно это изменение в модели можно отразить в виде нескольких конкретных подклассов, наследующих суперкласс `BillingDetails`: `CreditCard`, `BankAccount` и т. д. Каждый из этих подклассов содержит немного отличающиеся данные (и определяет совершенно разную функциональность для работы с этими данными). Диаграмма классов UML на рис. 1.3 демонстрирует эту модель.

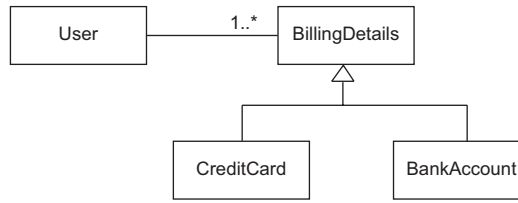

Рис. 1.3 ❖ Применение наследования для различных способов оплаты

Какие изменения требуется внести для поддержки обновленной структуры Java-классов? Требуется ли создавать таблицу `CREDITCARD`, *расширяющую* `BILLINGDETAILS`? Системы баз данных SQL, как правило, не поддерживают табличного наследования (или даже наследования типов), а используют нестандартный синтаксис и могут подвергать нас проблемам целостности данных (ограничивая поддержку целостности для представлений, допускающих обновление).

Наследование – не единственная трудность. Добавив наследование, мы также добавили в модель *полиморфизм*.

У класса `User` есть ассоциация с суперклассом `BillingDetails` – это *полиморфная ассоциация*. Во время выполнения экземпляр `User` может ссылаться на экземпляр любого из подклассов `BillingDetails`. Также хотелось бы иметь возможность создания *полиморфных запросов*, ссылающихся на класс `BillingDetails`, и чтобы запрос возвращал экземпляры подклассов.

В базах данных SQL нет способа (по крайней мере, стандартного) представления полиморфной ассоциации. Ограничение внешнего ключа ссылается только на одну таблицу; нет простого способа определить внешний ключ, ссылающийся на несколько таблиц. Для обеспечения такого типа целостности придется создать процедурное ограничение.

Из-за несоответствия подтипов производные структуры должны сохраняться в базе данных SQL, не предоставляющей механизмов наследования. В главе 6 мы обсудим, как ORM-системы, такие как Hibernate, решают проблему хранения иерархии классов в таблице (в таблицах) базы данных SQL и как может быть реализовано полиморфное поведение. К счастью, данная проблема хорошо изучена, и большинство решений поддерживает практически одинаковую функциональность.

Следующим аспектом объектно-реляционного несоответствия является проблема *идентичности объектов*. Вы, возможно, заметили, что в нашем примере мы

сделали столбец USERNAME таблицы USERS первичным ключом. Был ли это хороший выбор? Как работать с идентичными объектами в Java?

1.2.3. Проблема идентичности

Проблема идентичности, на первый взгляд, кажется неочевидной, но вы часто будете сталкиваться с ней в растущих и развивающихся системах электронной коммерции, например когда требуется проверить идентичность двух экземпляров. Существуют три подхода к решению этой проблемы: два – на стороне Java и один – в базе данных SQL. Как и следовало ожидать, нужно приложить усилия, чтобы они работали вместе.

Java определяет два различных понятия тождественности:

- идентичность экземпляров (грубо говоря, совпадение адресов в памяти; проверяется как a == b);
- равенство экземпляров, определяемое методом equals() (также называется *равенством по значению*).

С другой стороны, идентичность записей в базе данных определяется сравнением значений первичного ключа. Как будет показано в разделе 10.1.2, ни equals(), ни оператор == не всегда эквивалентны сравнению значений первичного ключа. Нередка ситуация, когда несколько неидентичных Java-объектов представляют одну и ту же запись в базе данных, например в параллельно выполняющихся потоках приложения. Кроме того, корректная реализация метода equals() для хранимого класса требует учета некоторых тонких нюансов и понимания, когда их стоит учитывать.

Воспользуемся нашим примером, чтобы продемонстрировать еще одну проблему, связанную с идентичностью в базе данных. Столбец USERNAME в таблице USERS играет роль первичного ключа. К сожалению, такая реализация усложняет смену имени пользователя – требуется обновить не только запись в таблице USERS, но и все значения внешнего ключа во многих строках таблицы BILLINGDETAILS. Далее в этой книге для решения данной задачи мы предложим использовать *суррогатный ключ*, когда не удается найти хороший естественный ключ. Также мы обсудим, что является хорошим первичным ключом. Столбец суррогатного ключа – это столбец первичного ключа, не имеющий значения для пользователя приложения, другими словами, это ключ, скрытый от пользователя приложения. Его единственная цель – идентифицировать данные внутри приложения.

К примеру, можно было бы поменять определения таблиц следующим образом:

```
create table USERS (
    ID bigint not null primary key,
    USERNAME varchar(15) not null unique,
    ...
);

create table BILLINGDETAILS (
    ID bigint not null primary key,
    ACCOUNT varchar(15) not null,
```

```
    BANKNAME varchar(255) not null,
    USER_ID bigint not null,
    foreign key (USER_ID) references USERS
);
```

Столбцы `ID` содержат сгенерированные системой значения. Раз эти столбцы были созданы исключительно ради самой модели данных, то каким образом (и нужно ли вообще) представлять их в модели Java? Мы обсудим этот вопрос в разделе 4.2 и найдем решение в ORM.

В контексте долговременного хранения данных идентичность тесно связана с тем, как система осуществляет кэширование и поддерживает транзакции. Различные решения используют разные стратегии, и это может сбивать с толку. Мы рассмотрим эти интересные темы и то, как они взаимосвязаны, в разделе 10.1.

На данном этапе прототип приложения для электронной коммерции уже выявил проблему несоответствия парадигм на примере детализации, подтипов и идентичности. Мы почти готовы двинуться дальше к другим частям приложения, но сначала обсудим важное понятие *ассоциаций*: как отображать и использовать отношения между сущностями. Является ли ограничение внешнего ключа в базе данных единственным, что для этого необходимо?

1.2.4. Проблемы, связанные с ассоциациями

Ассоциации в предметной модели представляют отношения между сущностями. Все классы – `User`, `Address` и `BillingDetails` – связаны между собой, но, в отличие от `Address`, класс `BillingDetails` стоит особняком. Экземпляры класса `BillingDetails` хранятся в отдельной таблице. Отображение ассоциаций и управление ассоциациями между сущностями являются ключевыми понятиями любого решения долговременного хранения объектов.

В объектно-ориентированных языках ассоциации представлены *объектными ссылками*; но в реляционном мире *столбец внешнего ключа* будет представлять ассоциацию при помощи дублирования значений этого ключа. Ограничение – это правило, гарантирующее целостность ассоциации. Между этими двумя способами существенные различия.

Объектные ссылки по своей природе обладают направленностью; ассоциация идет от одного экземпляра к другому. Они – указатели. Если ассоциация должна быть двунаправленной, следует определить ее *дважды*, по одному разу в каждом из ассоциированных классов. Вы это уже видели в классах предметной модели:

```
public class User {
    Set billingDetails;
}

public class BillingDetails {
    User user;
}
```

Навигация в конкретном направлении не имеет смысла для реляционной модели данных, потому что можно создавать произвольные ассоциации при помощи

операций *соединения* и *проекции*. Основная сложность в том, чтобы отобразить совершенно открытую модель данных, которая не зависит от приложения, работающего с данными, на зависимую от приложения навигационную модель – ограниченное представление ассоциаций для конкретного приложения.

Ассоциации в Java могут иметь вид *многие ко многим*. Классы, например, могли быть определены следующим образом:

```
public class User {
    Set billingDetails;
}

public class BillingDetails {
    Set users;
}
```

Но объявление внешнего ключа в таблице `BILLINGDETAILS` является ассоциацией *многие к одному*: каждый банковский счет привязан к конкретному пользователю. Каждый пользователь может иметь несколько банковских счетов.

Чтобы выразить ассоциацию *многие ко многим* в базе данных SQL, придется создать дополнительную таблицу, также называемую *таблицей ссылок* (link table). В большинстве случаев эта таблица отсутствует в предметной модели. Если для данного примера представить отношение между пользователем и платежными реквизитами как *многие ко многим*, таблицу ссылок можно определить следующим образом:

```
create table USER_BILLINGDETAILS (
    USER_ID bigint,
    BILLINGDETAILS_ID bigint,
    primary key (USER_ID, BILLINGDETAILS_ID),
    foreign key (USER_ID) references USERS,
    foreign key (BILLINGDETAILS_ID) references BILLINGDETAILS
);
```

Вам больше не нужен столбец внешнего ключа `USER_ID` и ограничение в таблице `BILLINGDETAILS`; связью между двумя сущностями теперь управляет эта дополнительная таблица. Мы обсудим ассоциации и отображение коллекций более подробно в главе 7.

Итак, проблемы, рассмотренные нами, считаются *структурными*: их можно заметить, рассматривая статическую картину системы. Возможно, наиболее трудной проблемой хранения объектов является *динамическая* проблема: порядок доступа к данным во время выполнения.

1.2.5. Проблемы навигации по данным

Существует фундаментальное различие между способами доступа к данным в Java и реляционной базе данных. Чтобы получить доступ к платежной информации пользователя в Java, вы вызываете `someUser.getBillingDetails().iterator().next()` или что-то подобное. Это наиболее естественный способ доступа к объект-

но-ориентированным данным, который обычно называется *обходом графа объектов*. Вы перемещаетесь от одного экземпляра к другому и даже перебираете коллекции, следуя за подготовленными указателями между классами. К сожалению, это не самый лучший способ получения информации из базы данных SQL.

Для улучшения производительности доступа к данным важно *уменьшить количество запросов к базе данных*. Самый очевидный способ достичь этого – уменьшить количество SQL-запросов. (Безусловно, за этим могут последовать другие, более сложные методы, такие как повсеместное кэширование.)

Таким образом, эффективный доступ к реляционным данным с помощью SQL обычно требует соединения интересующих нас таблиц. Количество соединяемых таблиц при извлечении данных определяет глубину графа объектов, доступных в памяти. Например, чтобы извлечь пользователя без его платежной информации, можно написать простой запрос:

```
select * from USERS u where u.ID = 123
```

С другой стороны, если требуется извлечь пользователя, а затем последовательно получить доступ к каждому связанному экземпляру `BillingDetails` (например, чтобы перечислить все счета пользователя), нужно составить другой запрос:

```
select * from USERS u
    left outer join BILLINGDETAILS bd
        on bd.USER_ID = u.ID
where u.ID = 123
```

Как видите, для эффективного использования соединения нужно *заранее* знать, какое подмножество графа объектов понадобится посетить! При этом важно проявлять осторожность: если вы извлечете слишком много данных (возможно, больше, чем могло бы понадобиться), вы потратите память на уровне приложения. Вы можете также нагрузить базу данных SQL большим декартовым произведением результирующих наборов. Представьте, что в одном запросе вы извлекаете не только пользователей и банковские счета, но и все заказы, оплаченные с каждого счета, товары в каждом заказе и т. д.

Каждая достойная применения система долговременного хранения объектов обеспечивает возможность извлечения данных ассоциированных экземпляров, только когда ассоциация действительно задействуется в Java-коде. Это называется *«отложенной загрузкой»*: данные извлекаются, только когда они действительно необходимы. Такой последовательный стиль доступа к данным крайне неэффективен в контексте баз данных SQL, поскольку требует выполнения одного выражения для каждого узла или коллекции графа объектов, по которому осуществляется обход. Это та страшная проблема *n + 1 запроса*.

Различие способов доступа к данным в Java и реляционных базах данных является, пожалуй, основным источником большинства проблем производительности в информационных системах, написанных на Java. Несмотря на огромное количество книг и статей, советующих использовать `StringBuffer` для конкатенации строк, для многих Java-программистов еще остается тайной, что следует избегать

проблем *декартова произведения* и *n + 1 запроса*. (Признайтесь: вы сейчас подумали, что `StringBuilder` был бы гораздо лучше, чем `StringBuffer`.)

Hibernate обладает сложной функциональностью для эффективного и прозрачного извлечения графов объектов из базы данных для последующего доступа к ним в приложении. Мы обсудим эту функциональность в главе 12.

У нас набрался целый список проблем объектно-реляционного несоответствия, и было бы очень затратно (как по времени, так и по усилиям) искать их решения, как вы, возможно, знаете по опыту. Потребовалась бы целая книга, чтобы подробно осветить эти вопросы и продемонстрировать практическое решение на основе ORM. Давайте начнем с обзора ORM, стандарта Java Persistence и проекта Hibernate.

1.3. ORM и JPA

Вкратце объектно-реляционное отображение – это автоматическое (и прозрачное) сохранение объекта из Java-приложения в таблицах базы данных SQL с использованием метаданных, описывающих отображение между классами приложения и схемой базы данных SQL. По сути, ORM работает за счет преобразования (двустороннего) данных из одного представления в другое. Прежде чем продолжить, вы должны понять, чего Hibernate *не сможет* сделать для вас.

Считается, что одним из преимуществ ORM является защита разработчика от неприятного языка SQL. При таком взгляде предполагается, что не следует ожидать от разработчиков объектно-ориентированных систем хорошего понимания SQL или реляционных баз данных и что SQL будет лишь действовать им на нервы. Мы же, напротив, считаем, что Java-разработчики должны быть достаточно хорошо знакомы с реляционными моделями данных и SQL (а также понимать их значение), чтобы работать с Hibernate. ORM является продвинутой технологией, которую используют разработчики, напряженно поработавшие над этими проблемами. Для эффективного использования Hibernate вы должны уметь читать и понимать выражения на языке SQL, которые генерирует фреймворк, и их влияние на производительность.

Давайте рассмотрим некоторые преимущества Hibernate.

- *Продуктивность* – Hibernate берет большую часть (больше, чем вы ожидаете) рутинной работы на себя, позволяя сконцентрироваться на проблеме предметной области. Не важно, какую стратегию разработки приложения вы предпочитаете – сверху вниз, начиная от предметной модели, или снизу вверх, начиная с существующей схемы базы данных, – Hibernate вместе с подходящими инструментами значительно *сократит время разработки*.
- *Простота сопровождения* – автоматизация объектно-реляционного отображения с Hibernate способствует уменьшению количества строк кода, делая систему более *понятной* и *удобной для рефакторинга*. Hibernate образует прослойку между предметной моделью и схемой SQL, предохраняя каждую модель от влияния незначительных изменений в другой.

- *Производительность* – несмотря на то что механизм хранения, реализованный вручную, может работать быстрее, так же как ассемблерный код будет выполняться быстрее Java-кода, автоматизированные решения, подобные Hibernate, позволяют использовать множество оптимизаций, работающих всегда. Одним из примеров может служить эффективное и легко настраиваемое кэширование на уровне приложения. Это означает, что разработчик сможет потратить больше энергии на то, чтобы вручную оптимизировать несколько оставшихся проблемных мест, вместо того чтобы предварительно оптимизировать все сразу.
- *Независимость от поставщика* – Hibernate может помочь снизить некоторые риски, связанные с зависимостью от поставщика. Даже если вы не планируете менять используемую СУБД, инструменты ORM, поддерживающие несколько различных СУБД, предоставляют вам *некоторый уровень переносимости*. Кроме того, независимость от СУБД обеспечивает такой способ разработки, когда инженеры используют *легковесную локальную базу данных*, а для тестирования и эксплуатации разворачивают приложение в другой системе.

Подход Hibernate к хранению данных был хорошо принят Java-разработчиками, и на его основе был разработан стандарт Java Persistence API.

Основные упрощения, внесенные в последние спецификации EJB и Java EE, коснулись JPA. Но мы должны сразу оговориться, что ни Java Persistence, ни Hibernate не ограничены окружением Java EE; они являются инструментами общего назначения для решения проблем долговременного хранения данных, которые могут использоваться любым приложением на Java (Scala, Groovy).

Спецификация JPA определяет следующее:

- способ определения метаданных отображений – как хранимые классы и их свойства соотносятся со схемой базы данных. JPA широко использует Java-аннотации в классах предметной модели, но вы можете определять отображения при помощи XML;
- API для основных CRUD-операций, производимых над экземплярами хранимых классов; наиболее известен класс `javax.persistence.EntityManager`, используемый для сохранения и загрузки данных;
- язык и API для создания запросов, использующих классы и их свойства. Этот язык называется Java Persistence Query Language (JPQL) и очень похож на SQL. Стандартизированный API позволяет программно создавать *запросы с критериями* без работы со строковыми значениями;
- порядок взаимодействия механизма хранения с транзакционными сущностями для сравнения состояний объектов (dirty checking), извлечения ассоциаций и выполнения прочих оптимизаций. Кроме того, в последней спецификации JPA рассмотрены основные стратегии кэширования.

Hibernate реализует JPA и поддерживает все стандартизированные отображения, запросы и программные интерфейсы.

1.4. Резюме

- Благодаря возможности долговременного хранения объектов отдельные объекты могут существовать дольше, чем процесс приложения; они могут быть помещены в хранилище данных, а позже восстановлены. Когда в роли хранилища данных выступает реляционная система управления базами данных, основанная на SQL, проявляются проблемы несоответствия объектной и реляционной парадигм. К примеру, граф объектов нельзя сохранить в таблицу базы данных непосредственно; прежде его необходимо разобрать, а затем сохранить в столбцы переносимых SQL-типов. Хорошим решением данной проблемы является объектно-реляционное отображение (ORM).
- Объектно-реляционное отображение не является идеальным решением для всех задач хранения; его цель – избавить разработчиков от 95% работы, связанной с хранением, например от написания сложных выражений SQL с большим количеством соединений таблиц и копирования полученных значений в объекты или графы объектов.
- Полноценное промежуточное программное обеспечение (middleware) для ORM может предоставить переносимость между базами данных, некоторые методы оптимизации, такие как кэширование, и некоторые другие практические аспекты, которые трудно реализовать вручную с помощью SQL и JDBC, имея при этом ограниченный запас времени.
- Однажды, возможно, появится решение лучше, чем ORM. Мы (как и многие другие), возможно, переосмыслим все, что нам известно о системах управления базами данных и их языках, о стандартах API доступа к хранимым данным, об интеграции приложений. Но превращение современных систем в подлинные реляционные системы с бесшовной поддержкой объектно-ориентированной парадигмы остается чистой фантазией. Мы не можем ждать, и нет никаких оснований полагать, что какие-то из проблем будут скоро решены (многомиллионная индустрия не очень гибкая). ORM в настоящее время является лучшим решением, которое сберегает время разработчиков, каждый день сталкивающихся с проблемой несоответствия объектной и реляционной парадигм.