Java Persistence API
и Hibernate

Java Persistence – механизм, помогающий обеспечить сохранность данных после завершения программы, что является главной чертой современных приложений. Hibernate – наиболее популярный инструмент Java для работы с базами данных, предоставляющим автоматическое прозрачное объектно-реляционное отображение, что значительно упрощает работу с SQL-базами данных в приложениях Java. Данная книга описывает разработку приложения с использованием Hibernate, связывая воедино сотни отдельных примеров. Вы сразу окунетесь в богатую моделями программирования среду Hibernate, которая основывается на отображениях, запросах, стратегиях выборки, транзакциях, диалогах, кэшировании и многом другом. Здесь вы найдете хорошо иллюстрированное обсуждение лучших методик проектирования баз данных и методов оптимизации. Авторы подробно описывают версию Hibernate 5, совместимую со стандартом Java Persistence 2.1.
Все примеры обновлены для последних версий спецификаций Hibernate и Java EE.

Что вошло в книгу:

- идея объектно-реляционного отображения;
- быстрая разработка приложений баз данных;
- исчерпывающее описание Hibernate и Java Persistence;
- интеграция Java Persistence с EJB, CDI, JSF и JAX-RS;
- непревзойденная широта и глубина охвата темы.

Кристиан Бауэр является членом команды разработчиков Hibernate, инструктором и консультантом.
Гэвин Кинг является основателем проекта Hibernate и членом экспертной группы Java Persistence (JSR 220).
Гэри Грегори является ведущим разработчиком программного обеспечения, работающим над серверными приложениями и интеграцией наследования.

Книга предполагает практическое знание Java.

«Самая полная книга о Hibernate! Одновременно и учебник, и руководство».
– Серхио Фернандес Гонсалес, Accenture Software

«Основной путеводитель по тонкостям Hibernate».
– Джоси Диас, OptumHealth

«Должна быть у каждого пользователя Hibernate».
– Стефан Хеффнер, SPIEGEL-Verlag Rudolf Augstein GmbH & Co. KG

ISBN 978-5-97060-180-8

Интернет-магазин: www.dmkpress.com
Книга – почтой: orders@alians-kniga.ru
Оптовая продажа: “Альянс-книга”
тел.(499)782-3889. books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Java Persistence API и Hibernate

Java Persistence API
и Hibernate

Кристиан Бауэр
Гэвин Кинг
Гэри Грегори

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
MANNING

Кристиан Бауэр, Гэвин Кинг, Гэри Грегори

Java Persistence API и Hibernate

Christian Bauer, Gavin King, Gary Gregory

Java Persistence with Hibernate

Second Edition

MANNING
SHELTER ISLAND

Кристиан Бауэр, Гэвин Кинг, Гэри Грегори

Java Persistence API и Hibernate

ДМК ПРЕСС
издательство
Москва, 2017

УДК 04.655.3Hibernate
ББК 32.973.2
Б29

Бауэр К., Кинг Г., Грегори Г.
Б29 Java Persistence API и Hibernate / пер. с англ. Д. А. Зинкевича; под науч. ред. А. Н. Киселева. – М.: ДМК Пресс, 2017. – 632 с.: ил.

ISBN 978-5-97060-180-8

Java Persistence – механизм, помогающий обеспечить сохранность данных после завершения программы, что является главной чертой современных приложений. Hibernate – наиболее популярный инструмент Java для работы с базами данных, предоставляющим автоматическое прозрачное объектно-реляционное отображение, что значительно упрощает работу с SQL-базами данных в приложениях Java. Данная книга описывает разработку приложения с использованием Hibernate, связывая воедино сотни отдельных примеров. Также вы найдете хорошо иллюстрированное обсуждение лучших методик проектирования баз данных и методов оптимизации.

Издание предназначено разработчикам, знакомым с языком Java.

УДК 04.655.3Hibernate
ББК 32.973.2

ISBN 978-1-617-29045-9 (анг.)
ISBN 978-5-97060-180-8 (рус.)

Александру, за то, что научил меня, как учить его.
– Гэри Грегори

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.

Благодарности

Мы не смогли бы написать эту книгу без помощи многих людей. Палак Матур (Palak Mathur) и Кристиан Альфано (Christian Alfano) проделали отличную работу, будучи техническими рецензентами нашей книги; спасибо вам за долгие часы, потраченные на редактирование наших «поломанных» примеров кода.

Мы также хотели бы поблагодарить наших рецензентов за потраченное время и неоценимую обратную связь в процессе разработки: Криса Бакара (Chris Bakar), Гаурава Бхардвая (Gaurav Bhardwaj), Якоба Босму (Jacob Bosma), Хосе Диаза (José Diaz), Марко Гамбини (Marco Gambini), Серхио Фернандеса Гонсалеса (Sergio Fernandez Gonzalez), Джерри Гуднафа (Jerry Goodnough), Джона Гриффина (John Griffin), Стефана Хеффнера (Stephan Heffner), Чеда Джонстона (Chad Johnston), Кристофа Мартини (Christophe Martini), Робби О'Коннора (Robby O'Connor), Антони Патрисио (Anthony Patricio) и Дениса Ванга (Denis Wang).

Издатель из Manning Марьян Бэйс (Marjan Bace) снова собрала отличную команду в Manning: Кристина Тэйлор (Christina Taylor) редактировала нашу сырую рукопись, превратив ее в настоящую книгу. Тиффани Тэйлор (Tiffany Taylor) нашла все опечатки, сделав книгу читаемой. Дотти Мариско (Dottie Marisco), ответственная за верстку, придала книге ее прекрасный вид. Мэри Пиргис (Mary Piergies) координировала и организовывала весь процесс. Мы благодарим всех вас за то, что работали с нами.

Наконец, отдельное спасибо Линде Демишель (Linda DeMichiel) за предисловие к первому изданию.

Гэри Грегори (Gary Gregory)

Я хотел бы поблагодарить моих родителей за то, что помогли мне начать это путешествие, дали мне прекрасное образование и свободу выбирать свой путь. Я бесконечно благодарен своей жене Лори и моему сыну Александру за то, что они предоставили мне время на завершение еще одного проекта – моей третьей книги.

В процессе работы я учился и сотрудничал с действительно выдающимися личностями, такими как Джордж Босворт (George Bosworth), Ли Брайзахер (Lee Breisacher), Кристофер Хэнсон (Christopher Hanson), Дебора Льюис (Deborah Lewis) и многими другими. Мой тесть, Бадди Мартин (Buddy Martin), заслуживает особого упоминания за то, что делился своей мудростью и проницательностью во время долгих бесед и рассказов историй, накопленных за те десятилетия, что он писал о спорте (вперед, «Аллигаторы»!). Я всегда нахожу вдохновение в му-

зыке, особенно в музыке следующих исполнителей: Wilco (Impossible Germany), Tom Waits (Blue Valentine), Donald Fagen (The Night-fly, A just machine to make big decisions/Programmed by fellows with compassion and vision), David Lindley и Баха. Наконец, я благодарю своего соавтора Кристиана Бауэра (Christian Bauer) за то, что делился со мной знаниями, и всех сотрудников издательства Manning за поддержку, профессионализм и доброжелательные отзывы.

Отдельное спасибо Тиффани Тэйлор (Tiffany Taylor) из Manning за то, что придала книге отличный вид. Дон Вэннер (Don Wanner), спасибо, точка.

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 механизмы 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.

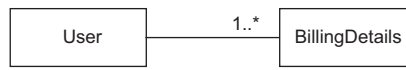


Рис. 1.1 ❖ Простая UML-диаграмма сущностей `User` и `Billing Details`

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;

// Методы доступа (чтения/записи), бизнес-методы и т. д.
}

public class BillingDetails {

String account;
String bankname;
User user;

// Методы доступа (чтения/записи), бизнес-методы и т. д.
}
```

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:

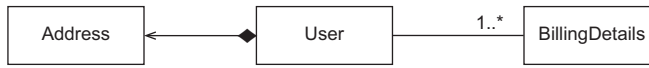


Рис. 1.2 ❖ У пользователя (User) есть адрес (Address)

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 было 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) Address в 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.

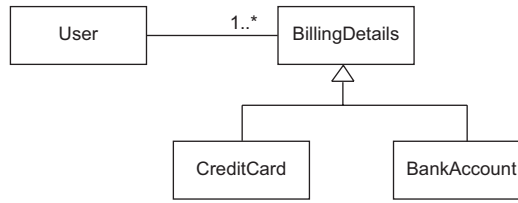


Рис. 1.3 ❖ Применение наследования для различных способов оплаты

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.

Глава 2

Создаем проект

В этой главе:
- обзор проектов с Hibernate;
- проект «Hello World» с Hibernate и Java Persistence;
- варианты конфигурации и интеграции.

В этой главе вы начнете работу с Hibernate и Java Persistence, используя пошаговый пример. Мы рассмотрим оба прикладных интерфейса и то, как извлечь выгоды от использования только Hibernate или стандартизированного JPA. Сначала мы предлагаем вашему вниманию обзор Hibernate на примере создания простейшего проекта «Hello World». Прежде чем начать программировать, вы должны решить, какие модули Hibernate будут использоваться в проекте.

2.1. Представляем Hibernate

Hibernate – это амбициозный проект, цель которого – дать полноценное решение проблемы управления хранимыми данными в Java. На сегодняшний день Hibernate – это не только служба ORM, но и набор инструментов управления данными, выходящих далеко за пределы ORM по своей функциональности.

Hibernate включает следующие проекты:

- *Hibernate ORM* – Hibernate ORM состоит из ядра, базовой службы хранения для баз данных SQL и оригинального API. Hibernate ORM является основой нескольких проектов и самым старым проектом Hibernate. Можно использовать один только фреймворк Hibernate ORM без привязки к конкретной платформе или среде выполнения с любым JDK. Он работает на любом сервере приложений Java EE/J2EE, в Swing-приложениях, в простых контейнерах сервлетов и т. д. Для его нормальной работы требуется только настроить источник данных;
- *Hibernate EntityManager* – реализация стандартного Java Persistence API, дополнительный модуль, устанавливаемый поверх Hibernate ORM. Вы в любой момент можете вернуться к использованию Hibernate, если требуется обычный программный интерфейс, или даже к JDBC Connection. Функциональность Hibernate является надмножеством функциональности JPA;

- *Hibernate Validator* – эталонная реализация спецификации Bean Validation (JSR 303). Независимо от других проектов Hibernate, она обеспечивает декларативную проверку классов предметной модели (или любых других классов);
- *Hibernate Envers* – целями Envers являются ведение журнала событий и хранение нескольких версий данных в базе данных SQL. Позволяет хранить историю изменения данных и событий подобно тому, как это делается в системах контроля версий, с которыми вы, возможно, уже знакомы, таких как Subversion и Git;
- *Hibernate Search* – поддерживает актуальный индекс предметных данных в базе данных Apache Lucene. Позволяет создавать запросы к этой базе данных, используя мощный и естественно интегрированный API. Многие проекты используют Hibernate Search в дополнение к Hibernate ORM для поддержки полнотекстового поиска. Если у вас имеется форма для поиска произвольного текста и вы хотели бы порадовать своих пользователей, используйте Hibernate Search. Данный проект мы не рассматриваем в этой книге; вы можете найти больше информации в книге «Hibernate Search in Action» (Bernard, 2008) Эммануэля Бернарда (Emmanuel Bernard);
- *Hibernate OGM* – самый последний проект Hibernate, реализация объектно-сеточного отображения (Object/Grid Mapping). Предоставляет поддержку JPA для NoSQL-решений, используя ядро движка Hibernate, но при этом отображая сущности в хранилища типа «ключ/значение», документные и графовые базы данных. Hibernate OGM не рассматривается в этой книге.

Давайте приступим к первому проекту с Hibernate и JPA.

2.2. «HELLO WORLD» и JPA

В этом разделе вы создадите свое первое Hibernate-приложение, которое сохраняет сообщение «Hello World» в базе данных, а затем извлекает его. Начнем с установки и настройки Hibernate.

Во всех примерах мы будем использовать инструмент сборки Apache Maven. Объявим зависимость от Hibernate:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.0.0.Final</version>
</dependency>
```

Модуль `hibernate-entitymanager` содержит транзитивные зависимости от других модулей, которые могут понадобиться, таких как `hibernate-core` и заглушки интерфейсов Java Persistence.

В JPA отправной точкой является *единица хранения* (persistence unit). Единица хранения объединяет отображение классов предметной области и соединение

с базой данных, а также содержит некоторые настройки. В каждом приложении есть как минимум одна единица хранения; в некоторых приложениях их несколько, если они взаимодействуют с несколькими (физическими или логическими) базами данных. Следовательно, первый шаг – настройка единицы хранения в конфигурации приложения.

2.2.1. Настройка единицы хранения

Обычно конфигурационный файл единицы хранения располагается в пути поиска классов (classpath), в файле *META-INF/persistence.xml*. Создайте следующий конфигурационный файл для приложения «Hello World»:

Файл: /model/src/main/resources/META-INF/persistence.xml

```
<persistence
        version="2.1"
        xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence_2_1.xsd">
    <persistence-unit name="HelloWorldPU">          ← ❶ Настройка единицы хранения
        <jta-data-source>myDS</jta-data-source>     ← ❷ Соединение с базой данных
        <class>org.jpwh.model.helloworld.Message</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>                                ← ❺ Набор свойств
            <property
                name="javax.persistence.schema-generation.database.action"
                value="drop-and-create"/>           ← ❻ Удалить схему и создать заново
            <property name="hibernate.format_sql" value="true"/>          ← ❼ Форматировать SQL
            <property name="hibernate.use_sql_comments" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

❹ Запретить поиск отображаемых классов

❸ Классы, хранимые в базе данных

❶ Файл *persistence.xml* содержит настройки как минимум одной единицы хранения; если их несколько, каждая должна иметь уникальное имя.

❷ Каждая единица хранения должна иметь соединение с базой данных. Здесь используется существующий экземпляр `java.sql.DataSource`. Hibernate отыщет источник данных по имени в каталоге JNDI во время запуска приложения.

❸ Единица хранения определяет хранимые (отображаемые) классы. Вы указываете их тут.

❹ Hibernate может искать отображаемые классы в каталоге классов и автоматически добавлять их в единицу хранения. Эта функция здесь отключена.

❺ Стандартные параметры или параметры конкретного поставщика можно определять как свойства единицы хранения. Любые стандартные свойства имеют префикс `javax.persistence`; свойства Hibernate имеют префикс `hibernate`.

❻ Движок JPA должен автоматически удалить и заново создать схему SQL в базе данных при загрузке. Это идеально для тестирования, когда важно иметь чистую базу данных при каждом запуске тестов.

❼ При выводе SQL в журнал позволяет форматировать код запросов и добавлять в него комментарии, чтобы вы понимали, почему Hibernate выполняет выражения SQL. Большинству приложений требуется пул соединений с базой данных с определенным размером и ограничениями, оптимизированными для конкретной среды. Также следует указать хост СУБД и учетные данные для подключения к базе данных.

Журналирование SQL

Каждое SQL-выражение, выполняемое фреймворком Hibernate, можно записать в журнал – незаменимый инструмент во время оптимизации. Для журналирования SQL-запросов установите значения свойств `hibernate.format_sql` и `hibernate.user_sql_comments` в значение `true` в файле *persistence.xml*. В этом случае Hibernate будет форматировать SQL-выражения с добавлением комментариев, описывающих причины выполнения этих запросов. Затем в настройках журналирования (которые зависят от выбранной реализации) в категориях `org.hibernate.SQL` и `org.hibernate.type.descriptor.sql.BasicBinder` установите самый подробный уровень вывода информации – отладочной. Благодаря этому вы сможете увидеть в журнале все SQL-выражения, выполняемые Hibernate, включая значения параметров в параметризованных запросах.

Для работы приложения «Hello World» управление соединениями с базой данных передается реализации Java Transaction API (JTA) – проекту *Bitronix* с открытым исходным кодом. Bitronix поддерживает пул соединений, используя управляемый `java.sql.DataSource` и стандартный `javax.transaction.UserTransaction` API в любом окружении Java SE. Bitronix включает эти объекты в каталог JNDI, и интерфейсы Hibernate будут автоматически использовать Bitronix посредством JNDI-поиска. Подробное описание установки Bitronix выходит за рамки этой книги; вы можете найти конфигурацию для наших примеров в `org.jpwh.env.TransactionManagerSetup`.

В приложении «Hello World» нам требуется организовать сохранение сообщений в базу данных и извлечение их оттуда. Для этого в Hibernate-приложениях определяются хранимые классы, которые затем отображаются в таблицы базы данных. Эти классы определяются на основе анализа предметной области; то есть они моделируют предметную область. Данный пример содержит один класс и его отображение.

Давайте посмотрим, как выглядит обычный хранимый класс, как создается отображение, и узнаем, что можно делать с экземплярами хранимых классов в Hibernate.

2.2.2. Хранимый класс

Цель данного примера – сохранить сообщение в базе данных, извлечь его обратно и показать. В приложении имеется один хранимый класс `Message`:

Файл: /model/src/main/java/org/jpwh/model/helloworld/Message.java

```
package org.jpwh.model.helloworld;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity                          ← ❶ Обязательная аннотация @Entity
public class Message {

    @Id                          ← ❷ Обязательная аннотация @Id
    @GeneratedValue              ← ❸ Значение ID будет генерироваться автоматически
    private Long id;

    private String text;         ← ❹ Отображение атрибутов

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

❶ Каждый класс хранимой сущности должен иметь хотя бы одну аннотацию `@Entity`. Hibernate автоматически отобразит этот класс в таблицу с названием `MESSAGE`.

❷ Каждый класс хранимой сущности должен иметь идентифицирующий атрибут, помеченный аннотацией `@Id`. Hibernate отобразит этот атрибут в столбец с названием `ID`.

❸ Значения идентификатора должны каким-то образом создаваться; эта аннотация настраивает автоматическое создание идентификаторов.

❹ Обычно атрибуты хранимого класса реализуются как приватные или защищенные свойства с общедоступными парами методов чтения/записи. Hibernate отобразит этот атрибут в столбец с названием `TEXT`.

Идентифицирующий атрибут позволяет приложению обращаться к идентификатору хранимой сущности в базе данных – значению первичного ключа. Если два экземпляра `Message` имеют одинаковое значение идентификатора, они представляют одну запись в базе данных.

В этом примере используется идентифицирующий атрибут типа `Long`, но это не является обязательным требованием. Как вы увидите позже, Hibernate позволяет использовать идентификаторы практически любого типа.

Вы наверняка заметили, что атрибут `text` класса `Message` имеет методы доступа к свойствам в стиле JavaBeans. Класс также имеет конструктор (по умолчанию) без параметров. Хранимые классы, которые мы используем в примерах, будут похожи на этот.

Экземпляры класса `Message` могут управляться (сохраняться) фреймворком Hibernate, но это не обязательно. Поскольку объект `Message` не реализует особых интерфейсов или классов, связанных с хранением, вы можете использовать его как обычный Java-класс:

```
Message msg = new Message();
msg.setText("Hello!");
System.out.println(msg.getText());
```

Может показаться, что мы здесь что-то мудрим; на самом деле мы хотим продемонстрировать важную особенность Hibernate, отличающую его от других решений хранения данных. Вы можете использовать хранимый класс в любом контексте исполнения – никакой специальный контейнер для этого не требуется.

Не обязательно даже использовать аннотации для отображения хранимого класса. Позже мы покажем вам другие способы отображения, такие как *orm_xml*, файл отображения для JPA, оригинальные файлы отображения *hbm_xml*, и расскажем, в каких случаях они оказываются предпочтительнее, чем аннотации в исходном коде.

Класс `Message` готов. Вы можете сохранять экземпляры в базе данных и писать запросы для их загрузки обратно в память приложения.

2.2.3. Сохранение и загрузка сообщений

Вам наверняка не терпится увидеть Hibernate в действии, так что давайте сохраним экземпляр `Message` в базе данных. Прежде всего для взаимодействий с базой данных вам понадобится `EntityManagerFactory`. Этот API представляет единицу хранения; в большинстве приложений поддерживается только один экземпляр `EntityManagerFactory` для одной единицы хранения:

Файл:/examples/src/est/java/org/jpwh/helloworld/HelloWorldJPA.java

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("HelloWorldPU");
```

В момент запуска приложение должно создать `EntityManagerFactory`; фабрика является потокобезопасной, и весь ваш код, взаимодействующий с базой данных, должен использовать ее.

Теперь вы сможете запустить транзакцию и сохранить экземпляр `Message`:

Файл:/examples/src/est/java/org/jpwh/helloworld/HelloWorldJPA.java

```
UserTransaction tx = TM.getUserTransaction();     ← ❶ Получаем доступ к UserTransaction
tx.begin();

EntityManager em = emf.createEntityManager();     ← ❷ Создаем EntityManager

Message message = new Message();     ← ❸ Создаем сообщение
message.setText("Hello World!");

em.persist(message);     ← ❹ Сохраняем сообщение

tx.commit();     ← ❺ Завершаем (commit) транзакцию
// INSERT into MESSAGE (ID, TEXT) values (1, 'Hello World!')

em.close();     ← ❻ Закрываем EntityManager
```

❶ Получаем доступ к стандартному API управления транзакциями – `UserTransaction` – и запускаем транзакцию в данном потоке выполнения.

❷ Начинаем новый сеанс работы с базой данных путем создания `EntityManager`. Этот объект послужит контекстом для всех операций с хранилищем.
❸ Создаем новый экземпляр отображаемого класса предметной модели `Message` и задаем его свойство `text`.
❹ Помещаем объект из памяти приложения (transient instance) в хранилище, делая его хранимым. Теперь Hibernate знает, что вы хотели бы сохранить эти данные, но он не обязательно обратится к базе данных в тот же момент.
❺ Завершаем транзакцию. Hibernate автоматически проверит контекст хранения и выполнит необходимую SQL-инструкцию `INSERT`.
❻ Если `EntityManager` создан вами, вы должны его закрыть.

Чтобы помочь вам разобраться в том, как работает Hibernate, мы будем показывать автоматически созданные и выполненные SQL-выражения в виде комментариев в исходном коде. Hibernate вставляет запись в таблицу `MESSAGE` с автоматически сгенерированным значением столбца первичного ключа `ID` и значением атрибута `TEXT`.

Вы можете загрузить эти данные позже, используя запрос к базе данных:

Файл:/examples/src/est/java/org/jpwh/helloworld/HelloWorldJPA.java

```
UserTransaction tx = TM.getUserTransaction();     ← ❶ Граница транзакции
tx.begin();

EntityManager em = emf.createEntityManager();

List<Message> messages =     ← ❷ Выполнение запроса
    em.createQuery("select m from Message m").getResultList();
// SELECT * from MESSAGE

assertEquals(messages.size(), 1);
assertEquals(messages.get(0).getText(), "Hello World!");

messages.get(0).setText("Take me to your leader!");     ← ❸ Меняем значение свойства

tx.commit();     ← ❹ Выполнит UPDATE
// UPDATE MESSAGE set TEXT = 'Take me to your leader!' where ID = 1

em.close();
```

❶ Каждое взаимодействие с базой данных должно осуществляться внутри явно заданных границ транзакции, даже если вы просто читаете данные.
❷ Выполняем запрос для извлечения всех сообщений из базы данных.
❸ Вы можете поменять значение свойства. Hibernate автоматически обнаружит это изменение, т. к. загруженный экземпляр `Message` остается связанным с контекстом хранения, куда он был загружен.
❹ В момент фиксации транзакции Hibernate проверит контекст хранения на предмет изменения состояния и автоматически выполнит SQL-запрос `UPDATE` для синхронизации состояния базы данных с объектом в памяти.

Язык запросов, который вы увидели в примере, – это не SQL, а Java Persistence Query Language (JPQL). В этом простейшем примере нет синтаксической разни-

цы, тем не менее `Message` в тексте запроса означает не имя таблицы базы данных, а имя хранимого класса. Если вы отобразите класс в другую таблицу, запрос будет продолжать работать.

Также стоить отметить, что Hibernate автоматически определяет изменение текста сообщения и сохраняет их в базе данных. Это – одна из особенностей JPA, автоматическое сравнение состояния объектов в действии. Больше нет необходимости явно требовать от диспетчера хранилища обновить базу данных, когда вы меняете состояние объекта в транзакции.

Вот вы и закончили создание своего первого приложения с Hibernate и JPA. Вы, возможно, заметили, что мы предпочитаем писать примеры в виде исполняемых тестов, используя утверждения (assertions) для проверки правильности результата каждой операции. Мы взяли все примеры для этой книги из тестового кода, чтобы вы (и мы) могли быть уверены, что все работает как надо. Но это, к сожалению, означает, что вам потребуется добавить несколько строк кода, чтобы создать `EntityManagerFactory` в тестовом окружении. Мы постарались сделать настройку тестов как можно более простой. Этот код вы найдете в `org.jpwh.env.JPASetup` и `org.jpwh.env.JPATest`; используйте его как отправную точку при написании собственных тестов.

Прежде чем начать работу над более жизненными примерами, давайте посмотрим, как осуществляется настройка Hibernate.

2.3. Оригинальная конфигурация Hibernate

Несмотря на то что основная (и обширная) настройка стандартизирована в JPA, вы не сможете получить доступа ко всем нюансам конфигурации Hibernate через *persistence.xml*. Отметим также, что большинство приложений, даже весьма сложных, не требует особых настроек и, таким образом, не использует API для начальной загрузки, показанный в этом разделе. Если вы не уверены, можете пропустить этот раздел, чтобы вернуться к нему позже, когда потребуется расширить адаптеры типов (type adapters) Hibernate, добавить нестандартные SQL-функции и т. д.

Оригинальным эквивалентом `EntityManagerFactory` из JPA является класс `org.hibernate.SessionFactory`. Как правило, он один во всем приложении; также он объединяет отображения классов с конфигурацией подключения к базе данных.

Оригинальный API для начальной загрузки Hibernate состоит из нескольких уровней, предоставляющих доступ к определенным аспектам конфигурации. В наиболее кратком виде построение `SessionFactory` выглядит следующим образом:

Файл:/examples/src/est/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
SessionFactory sessionFactory = new MetadataSources(
    new StandardServiceRegistryBuilder()
        .configure("hibernate.cfg.xml").build()
).buildMetadata().buildSessionFactory();
```

Этот код загрузит все настройки из конфигурационного файла Hibernate. Если у вас есть существующий проект, возможно, этот файл уже находится в пути поиска классов. Подобно *persistence.xml*, этот конфигурационный файл, содержит детали подключения к базе данных, а также список хранимых классов и других свойств конфигурации.

Давайте проанализируем этот фрагмент кода и рассмотрим API более детально. Сначала создадим экземпляр `ServiceRegistry`:

Файл:/examples/src/est/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
StandardServiceRegistryBuilder serviceRegistryBuilder =     ← ❶ Конструктор
    new StandardServiceRegistryBuilder();

serviceRegistryBuilder     ← ❷ Конфигурация реестра служб
    .applySetting("hibernate.connection.datasource", "myDS")
    .applySetting("hibernate.format_sql", "true")
    .applySetting("hibernate.use_sql_comments", "true")
    .applySetting("hibernate.hbm2ddl.auto", "create-drop");
ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
```

❶ Этот конструктор поможет создать неизменяемый реестр служб посредством вызова цепочки методов.
❷ Настраиваем реестр служб, применяя настройки.

Если вы хотите вынести конфигурацию реестра служб в отдельный файл, можете загружать настройки из файла свойств в пути поиска классов при помощи `StandardServiceRegistryBuilder#load-Properties(file)`.

Построив неизменяемый экземпляр `ServiceRegistry`, вы можете перейти к следующему этапу: сообщить Hibernate, какие из хранимых классов являются частью метаданных отображения. Источники метаданных настраиваются следующим образом:

Файл:/examples/src/est/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
MetadataSources metadataSources = new MetadataSources(serviceRegistry);

metadataSources.addAnnotatedClass(
org.jpwh.model.helloworld.Message.class
);

// Добавить файлы отображения hbm.xml
// metadataSources.addFile(...);

// Прочитать все файлы отображения hbm.xml из сборки JAR
// metadataSources.addJar(...)

MetadataBuilder metadataBuilder = metadataSources.getMetadataBuilder();
```

❷ Добавляем хранимые классы в источники метаданных
❶ Требуется реестр служб

❶ Этот конструктор помогает создать неизменяемый реестр служб с использованием цепочки вызовов методов.
❷ Конфигурирование реестра служб применением настроек.

В прикладном программном интерфейсе `MetadataSources` имеется множество методов для добавления источников отображений; за дополнительной инфор-

мацией обращайтесь к документации JavaDoc. Следующий этап процедуры начальной загрузки – построение всех метаданных, нужных фреймворку Hibernate, с помощью экземпляра `MetadataBuilder`, полученного из источников метаданных.

После этого можно запросить метаданные для программного взаимодействия с конфигурацией Hibernate или продолжить и построить `SessionFactory`:

Файл:/examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
Metadata metadata = metadataBuilder.build();

assertEquals(metadata.getEntityBindings().size(), 1);

SessionFactory sessionFactory = metadata.buildSessionFactory();
```

Создание EntityManagerFactory с помощью SessionFactory

На момент написания книги Hibernate не имел удобного API для программного создания `EntityManagerFactory`. Для этой цели можно использовать внутренний API: класс `org.hibernate.jpa.internal.EntityManagerFactoryImpl` имеет конструктор, принимающий экземпляр `SessionFactory`.

Давайте убедимся, что данная конфигурация работает, сохранив и загрузив сообщение с помощью оригинального Hibernate-эквивалента класса `EntityManager` – `org.hibernate.Session`. Создать экземпляр `Session` можно при помощи `SessionFactory`, кроме того, сеанс должен закрываться так же, как `EntityManager`.

Или можно использовать другую особенность Hibernate и предоставить фреймворку самому позаботиться о закрытии `Session` с помощью `SessionFactory#getCurrentSession()`:

Файл:/examples/src/ test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
UserTransaction tx = TM.getUserTransaction();    ← ❶ Получаем доступ к UserTransaction
tx.begin();

Session session =
    sessionFactory.getCurrentSession();    ← ❷ Получаем org.hibernate.Session

Message message = new Message();
message.setText("Hello World!");

session.persist(message);    ← ❸ Прикладные интерфейсы JPA и Hibernate схожи

tx.commit();    ← ❹ Фиксируем транзакцию
// INSERT into MESSAGE (ID, TEXT) values (1, 'Hello World!')
```

❶ Получаем доступ к стандартному программному интерфейсу управления транзакциями – `UserTransaction` – и запускаем транзакцию в текущем потоке выполнения.

❷ Когда бы вы ни вызывали `getCurrentSession()` в том же потоке, вы всегда будете получать тот же экземпляр `org.hibernate.Session`. Он автоматически связывается с текущей транзакцией и закрывается автоматически при фиксации или откате транзакции.

❸ Прикладной интерфейс Hibernate очень похож на стандартный Java Persistence API, и большинство методов называется одинаково.
❹ Hibernate синхронизирует сеанс с базой данных и автоматически закрывает «текущий» сеанс в момент фиксации связанной транзакции.

Использование текущего сеанса помогает сократить код:

Файл:/examples/src/test/java/org/jpwh/helloworld/HelloWorldHibernate.java

```
UserTransaction tx = TM.getUserTransaction();
tx.begin();

List<Message> messages =     ← ❶ Запрос на основе критериев
    sessionFactory.getCurrentSession().createCriteria(
        Message.class
    ).list();
// SELECT * from MESSAGE

assertEquals(messages.size(), 1);
assertEquals(messages.get(0).getText(), "Hello World!");

tx.commit();
```

❶ Запрос на основе критериев Hibernate – это типобезопасный способ программного составления запросов, автоматически преобразуемых в SQL.

Большинство примеров в этой книге не использует классов `SessionFactory` и `Session`. Время от времени, когда определенная функциональность будет доступна только в Hibernate, мы будем демонстрировать, как доставать (`unwrap()`) оригинальные интерфейсы из стандартных API.

2.4. Резюме

- Вы завершили свой первый JPA-проект.
- Вы написали первый хранимый класс и его отображение с помощью аннотаций.
- Вы узнали, как настроить и выполнить начальную загрузку единицы хранения, как создать точку доступа `EntityManagerFactory`. Затем вы использовали `EntityManager` для взаимодействия с базой данных, сохраняя и загружая экземпляры хранимых классов предметной модели.
- Мы обсудили некоторые более продвинутые возможности настройки Hibernate, а также эквивалентные базовые API, предоставляемые Hibernate: `SessionFactory` и `Session`.

Глава 3

Модели предметной области и метаданные

В этой главе:
- учебное приложение CaveatEmptor;
- реализация предметной модели;
- типы метаданных объектно-реляционного отображения.

Пример «Hello World» из предыдущей главы познакомил вас с Hibernate; очевидно, он не особо полезен для понимания требований к реальным приложениям со сложными моделями данных. До конца книги мы будем использовать более сложное учебное приложение – CaveatEmptor, систему онлайн-аукциона – для демонстрации возможностей Hibernate и Java Persistence («caveat emptor» означает по-латыни «Пусть покупатель будет бдителен»).

Главные нововведения в JPA 2

- Реализации JPA теперь автоматически интегрируются с реализациями Bean Validation. В момент сохранения данных реализация автоматически проверяет ограничения хранимых классов.
- Добавлен класс `Metamodel`. Вы можете получать доступ (но, к сожалению, не изменять) к именам, свойствам и метаданным отображения классов в единице хранения.

Начнем обзор приложения со знакомства с его многоуровневой архитектурой. После этого вы узнаете, как выявлять бизнес-сущности предметной области. Создадите концептуальную модель этих сущностей и их атрибутов, называемую *моделью предметной области*, и реализуете ее на Java в виде хранимых классов. Мы потратим немного времени, чтобы выяснить, как эти классы должны выглядеть

и какое место они занимают в типичной архитектуре многоуровневого приложения. Мы также рассмотрим возможности долговременного хранения классов и то, как эта особенность влияет на проектирование и реализацию. Мы добавим поддержку *Bean Validation* для автоматической проверки целостности данных предметной области, относящихся не только к хранимой информации, но и ко всей бизнес-логике.

Затем мы исследуем типы метаданных отображений, помогающих сообщить фреймворку Hibernate, как хранимые классы и их свойства соотносятся с таблицами базы данных и столбцами. Это делается так же просто, как добавление аннотаций прямо в исходный код Java-классов или создание XML-документов, которые впоследствии будут развернуты вместе с откомпилированными Java-классами и к которым Hibernate сможет получить доступ во время выполнения. Прочитав эту главу, вы узнаете, как в по-настоящему сложных проектах определять те части предметной области, которые связаны с хранением, и какой тип метаданных предпочтительнее в той или иной ситуации. Приступим к созданию приложения.

3.1. Учебное приложение CaveatEmptor

Пример CaveatEmptor – это приложение онлайн-аукциона, демонстрирующее приемы работы с ORM и возможности Hibernate. Вы можете загрузить исходный код приложения по адресу: http://www.jpwh.org. В этой книге мы не будем уделять большого внимания пользовательскому интерфейсу (это может быть веб- или «толстый» клиент), а сосредоточимся на реализации доступа к данным. В том случае, когда проектные решения, касающиеся доступа к данным, будут иметь последствия для пользовательского интерфейса, мы естественно будем обращать внимание на оба аспекта.

Чтобы понять проблемы проектирования, связанные с ORM, притворимся, что приложения CaveatEmptor еще не существует и вы создаете его с нуля. Начнем с рассмотрения архитектуры.

3.1.1. Многоуровневая архитектура

В любом нетривиальном приложении обычно имеет смысл организовать классы согласно областям ответственности. Долговременное хранение – одна область ответственности; другие включают представление, рабочий процесс и бизнес-логику. Типичная объектно-ориентированная архитектура состоит из уровней, представляющих области ответственности.

Сквозные задачи

Кроме всех прочих, существуют еще так называемые *сквозные задачи*, которые могут иметь реализацию, общую для всех приложений, например в коде фреймворка. К типичным сквозным задачам относятся журналирование, авторизация и разграничение транзакций.

Многоуровневая архитектура определяет интерфейсы, используемые реализациями различных областей ответственности, и позволяет вносить изменения в реализацию каждого уровня без серьезного нарушения работы других. Подобное разделение определяет типы зависимостей между уровнями. Многоуровневая архитектура должна подчиняться следующим правилам:

- передача информации осуществляется сверху вниз. Каждый уровень зависит только от интерфейса того уровня, который находится непосредственно под ним;
- каждый уровень ничего не знает о других уровнях, кроме нижележащего.

В различных системах области ответственности группируются по-разному, определяя различные уровни. Типичная проверенная временем высокоуровневая архитектура приложения использует три уровня: представления, бизнес-логики и хранения, как показано на рис. 3.1.

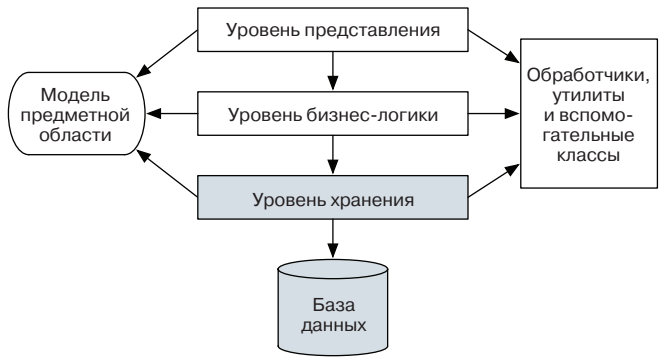


Рис. 3.1 ❖ Уровень хранения является базовым в многоуровневой архитектуре

- *Уровень представления* – логика пользовательского интерфейса находится на самом верху. Код, отвечающий за представление, управление страницей и навигацию, находится в уровне представления. Код пользовательского интерфейса может обращаться напрямую к бизнес-сущностям общей *модели предметной области* и отображать их на экране вместе с элементами управления для выполнения различных действий. В некоторых архитектурах экземпляры бизнес-сущностей могут быть недоступны коду пользовательского интерфейса напрямую: например, когда уровень представления работает на другом компьютере. В таких случаях уровню представления может потребоваться собственная специальная модель передачи данных, представляющая лишь подмножество модели предметной области, подходящей для передачи.
- *Уровень бизнес-логики* – конкретные формы этого уровня сильно различаются в зависимости от приложения. Считается, что уровень бизнес-логики отвечает за реализацию любых бизнес-правил или системных требований,

которые воспринимаются пользователями как относящиеся к предметной области. Этот уровень обычно содержит некий управляющий компонент, который знает, когда и какое бизнес-правило применять. В одних системах этот уровень имеет свое внутреннее представление сущностей предметной области. В других – он зависит от реализации предметной модели, общей для всех уровней приложения.

- *Уровень хранения* – уровень длительного хранения данных состоит из набора классов и компонентов, ответственных за сохранение и извлечение данных из одного или нескольких хранилищ. Этому уровню требуется модель сущностей предметной области, состояния которых предполагается хранить. Уровень хранения – то место, где больше всего используются JPA и Hibernate.
- *База данных* – как правило, является внешней и используется несколькими приложениями. Если используется база данных SQL, у нее есть схема и, возможно, хранимые процедуры для выполнения бизнес-логики в непосредственной близости к данным.
- *Вспомогательные и служебные классы* – в каждом приложении имеется набор инфраструктурных вспомогательных и служебных классов, которые используются во всех уровнях приложения (как, например, класс `Exception` для обработки ошибок). Эти общие инфраструктурные элементы не образуют отдельного уровня, т. к. для них не выполняются правила зависимости между уровнями в многоуровневой архитектуре.

Теперь, после знакомства с высокоуровневой архитектурой, можно сосредоточиться на решении проблем предметной области.

3.1.2. Анализ предметной области

На следующем этапе вы, совместно с экспертами в предметной области, должны проанализировать проблемы, которые предстоит решать вашему программному обеспечению, а также определить основные сущности и способы взаимодействий между ними. Главный мотив проведения анализа и проектирования предметной модели состоит в выявлении сути предметной области для целей разработки приложения.

Сущности, как правило, являются понятиями, знакомыми пользователям системы: платеж, клиент, заказ, товар, предложение цены на аукционе и т. д. Некоторые сущности могут представлять менее конкретные абстракции, такие как алгоритм формирования цен; но даже такие сущности понятны пользователям. Вы можете найти все эти сущности в концептуальном представлении бизнеса, которое иногда называют *бизнес-моделью*.

На основе этой бизнес-модели инженеры и архитекторы объектно-ориентированного ПО создают объектно-ориентированную модель, но пока лишь на концептуальном уровне (без Java-кода). Такая модель может быть столь же простой, как мысленный образ, находящийся лишь в голове разработчика, или столь же подробной, как UML-диаграмма класса. На рис. 3.2 показана простая модель, представленная с помощью UML.

Рис. 3.2 ❖ Диаграмма классов модели типичного онлайн-аукциона

Эта модель содержит сущности, которые вы обязательно найдете в любой системе электронной торговли: категория, товар и пользователь. Эта предметная модель представляет все сущности и их отношения (и, возможно, атрибуты). Мы называем такой вид объектно-ориентированной модели, включающей только наиболее важные для пользователя сущности предметной области, *моделью предметной области* (или *предметной моделью*). Она является абстрактным представлением реального мира.

Вместо объектно-ориентированной модели инженеры и архитекторы могут начать проектирование приложения с модели данных (возможно, представленной диаграммами «сущность–связь»). Мы обычно говорим, что по отношению к механизму хранения между двумя моделями нет большой разницы; они просто являются разными отправными точками. В конечном итоге выбор языка моделирования играет второстепенную роль; больше всего мы заинтересованы в структурах и отношениях бизнес-сущностей. Нас заботят правила, которые необходимо применять для поддержания целостности данных, и процедуры, манипулирующие этими данными.

В следующем разделе мы завершим анализ предметной области приложения CaveatEmptor. Полученная модель предметной области станет центральной темой этой книги.

3.1.3. Предметная модель приложения CaveatEmptor

Сайт CaveatEmptor продает с аукциона различные виды товаров – от электронного оборудования до билетов на самолет. Аукцион проходит согласно английской аукционной стратегии: пользователи продолжают предлагать цену за товар, пока не закончится торг, после чего побеждает предложивший наибольшую цену.

В любом магазине товары делятся на категории и объединяются с похожими товарами в отделах и на полках. Каталог аукциона требует некоторой иерархии категорий товаров, чтобы покупатель мог просматривать эти категории или произвольно искать по категории или атрибутам товара. Списки товаров появляются в браузере категорий и на страницах с результатами поиска. Выбор товара в списке открывает для покупателя его детальное представление, где у товара могут быть прикрепленные изображения.

Аукцион состоит из последовательности предложений цены и одного победившего предложения. Данные о пользователях включают имя, адрес и платежную информацию.

Результат этого анализа – высокоуровневая схема предметной модели – показан на рис. 3.3. Давайте коротко обсудим некоторые интересные особенности этой модели.

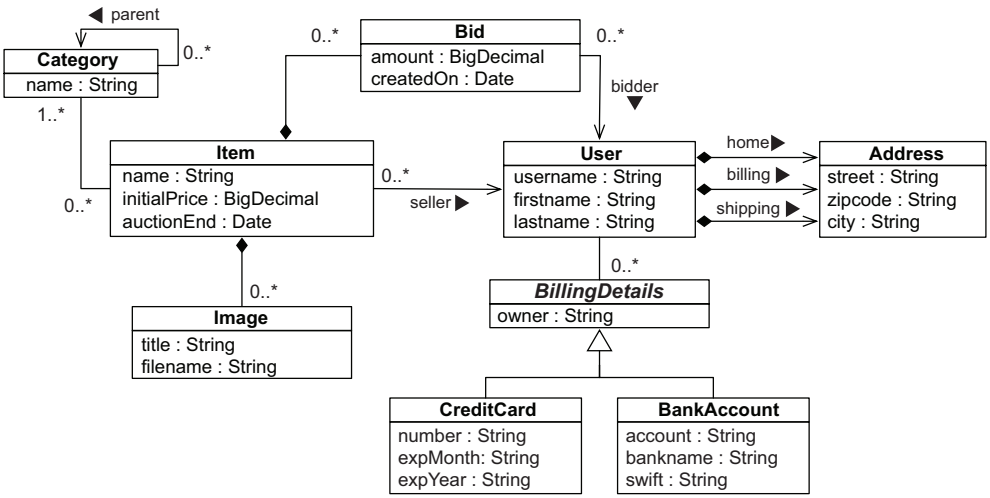


Рис. 3.3 ❖ Хранимые классы предметной модели CaveatEmptor и их отношения

Каждый товар может быть продан только один раз, поэтому нет смысла делать сущность, представляющую товар (`Item`), отличной от любой другой сущности на аукционе. Вместо этого у вас будет одна сущность для аукционного товара – `Item`. Соответственно, предложение цены (`Bid`) будет иметь прямую ассоциацию с товаром. Информация об адресе моделируется как отдельный класс `Address`; пользователь может иметь три адреса: домашний (`home`), платежный (`billing`) и доставки (`shipping`). Пользователь может иметь несколько вариантов платежной информации (`BillingDetails`). Подклассы абстрактного класса представляют различные платежные стратегии (допуская будущие расширения).

Категория (`Category`) может быть вложена внутрь другой категории. Рекурсивная ассоциация сущности с самой собой (`parent`) выражает это отношение. Отметим, что категория может иметь несколько дочерних категорий, но лишь одну родительскую. Каждый товар относится, по меньшей мере, к одной категории.

Эта схема не является *полноценной* предметной моделью. Она содержит лишь классы, которые требуют длительного хранения. Нам определенно необходимо сохранять и загружать экземпляры классов `Category`, `Item`, `User` и т. д. Мы немного упростили высокоуровневую схему, но можем добавить дополнительные классы позже или внести незначительные изменения, когда нам потребуются более сложные примеры.

Сущности предметной модели, безусловно, должны инкапсулировать состояние и поведение. К примеру, сущность `User` должна определять имя и адрес клиента, *а также* логику вычисления стоимости доставки товаров (для этого конкретного клиента).

В модели предметной области также могут быть классы, экземпляры которых существуют лишь в памяти приложения в момент выполнения. Рассмотрим класс

`WinningBidStrategy`, инкапсулирующий логику победы на аукционе предложившего наибольшую цену. Он может вызываться уровнем бизнес-логики (контроллером) во время проверки состояния аукциона. В какой-то момент, возможно, вам придется выяснить, как вычисляется налог на проданный товар или как система может разрешить создание новой учетной записи пользователя. Мы не считаем подобную бизнес-логику или поведение предметной модели неважными; скорее, эта функциональность является ортогональной по отношению к хранению данных.

Теперь, когда у вас есть проект приложения (пусть и примитивный) с предметной моделью, следующим шагом будет его реализация на Java.

ORM без предметной модели

Хранение объектов с полноценным объектно-реляционным отображением больше подходит приложениям с насыщенной предметной моделью. Если ваше приложение не реализует сложной бизнес-логики или сложных взаимодействий между сущностями (или если сущностей очень немного), предметная модель может не понадобиться. Для большинства простых, а иногда и не слишком простых, решений отлично подходят таблично-ориентированные (table-oriented) решения, когда приложение проектируется вокруг схемы базы данных, а не вокруг объектно-ориентированной предметной модели, и логика реализована на стороне базы данных (в виде хранимых процедур). Другой аспект, требующий рассмотрения, – это кривая обучения: освоив фреймворк Hibernate, вы будете использовать его в любых приложениях, даже таких простых, которые генерируют SQL-запросы и отображают результаты. Если вы еще только изучаете ORM, реализация простейшего примера использования может не оправдать ваших затрат.

3.2. Реализация предметной модели

Начнем с проблемы, с которой сталкивается любая реализация: разделение ответственности. Как правило, реализация предметной модели – это центральный организующий компонент; она повторно используется всякий раз, когда в приложение добавляется новая функциональность. Поэтому будьте готовы потрудиться, чтобы задачи, не касающиеся предметной области, не просочились в реализацию предметной модели.

3.2.1. Предотвращение утечек функциональности

Когда такие области ответственности, как хранение данных, управление транзакциями или авторизация, просачиваются в классы предметной модели, – это пример утечки функциональности (leakage of concerns). Реализация предметной модели является настолько важным участком кода, что она должна быть ортогональна любым Java API. К примеру, код предметной модели не должен выполнять JNDI-запросов или обращаться к базе данных посредством JDBC API ни напрямую, ни с использованием промежуточных абстракций. Это позволяет повторно использовать классы предметной модели практически повсеместно:

- уровень представления может получать доступ к экземплярам и атрибутам сущностей предметной модели для вывода представлений;
- контроллеры уровня бизнес-логики также могут обращаться к состоянию сущностей предметной модели и вызывать методы сущностей для выполнения бизнес-логики;
- уровень хранения может сохранять экземпляры в базе данных и извлекать их из нее.

Более того, предотвращение утечки функциональности позволяет с легкостью проводить модульное тестирование предметной модели без необходимости использовать конкретную среду выполнения или контейнер или имитировать какие-либо зависимости от служб. Можно создавать модульные тесты, проверяющие корректность поведения классов модели, без использования специального тестового фреймворка (мы не говорим сейчас о тестировании аспектов загрузки из базы данных или сохранении в базу данных, но о таких аспектах, как расчет стоимости доставки и налога).

Стандарт JavaEE решает проблему утечки функциональности при помощи метаданных в виде аннотаций или внешних XML-дескрипторов. Такой подход позволяет контейнеру среды выполнения реализовывать в общем виде некоторую заранее определенную сквозную функциональность: безопасность, многопоточность, хранение данных, поддержку транзакций и удаленные взаимодействия – путем перехвата вызовов к компонентам приложения. Hibernate не является ни средой выполнения JavaEE, ни сервером приложений. Этот проект является реализацией лишь одной спецификации из состава JavaEE – JPA – и решением лишь для одной области ответственности – хранения данных.

JPA определяет *класс сущности* как первичный программный артефакт. Эта модель программирования поддерживает прозрачный подход к хранению данных, а поставщик JPA, такой как Hibernate, предоставляет также возможность автоматизации.

3.2.2. Прозрачность сохранения и его автоматизация

Мы используем слово *прозрачность*, чтобы подчеркнуть полное разделение ответственности между хранимыми классами предметной модели и уровнем хранения. Хранимые классы ничего не знают о механизме хранения данных и не зависят от него. Под словом *автоматизация* мы подразумеваем способность решения хранения данных (включающего аннотированные классы, уровень хранения и механизм хранения) избавлять вас от решения низкоуровневых задач – написания большинства SQL-выражений и взаимодействий с JDBC API.

Класс `Item` предметной модели CaveatEmptor, например, не должен иметь никаких зависимостей времени выполнения от какого-либо прикладного программного интерфейса Java Persistence или Hibernate. Более того:

- JPA не требует, чтобы хранимый класс наследовал какой-то специальный суперкласс или реализовал интерфейс. Также не существует специальных классов для реализации атрибутов и ассоциаций (безусловно, возможность использования обоих вариантов всегда присутствует);

- хранимые классы можно повторно использовать вне контекста хранения, например в модульных тестах или на уровне представления. Вы можете создавать экземпляры в любой среде выполнения, используя обычный оператор `new`, сохраняя возможности для тестирования и повторного использования;
- в системе с прозрачной поддержкой хранения экземпляры сущностей понятия не имеют о хранилище данных; они не должны даже понимать, что могут сохраняться или извлекаться. JPA передает ответственность за хранение универсальному API диспетчера механизмов хранения;
- следовательно, большая часть вашего кода и, конечно, бизнес-логики не должна беспокоиться о текущем состоянии экземпляров сущностей предметной модели ни в каком потоке выполнения.

Мы рассматриваем прозрачность как требование, потому что она упрощает разработку и сопровождение приложений. Прозрачность должна быть основной целью любого ORM. Очевидно, что ни одно автоматическое решение не является полностью прозрачным: каждый уровень автоматизации, включая JPA и Hibernate, накладывает некоторые ограничения на хранимые классы. Например, JPA требует, чтобы атрибуты-коллекции имели типы интерфейсов, таких как `java.util.Set` или `java.util.List`, а не конкретных реализаций, таких как `java.util.HashSet` (вообще-то, это хорошая практика). Также класс сущности JPA должен иметь особый атрибут – *идентификатор в базе данных* (что является скорее удобством, чем ограничением).

Теперь вы понимаете, почему механизм хранения должен иметь ограниченное влияние на реализацию модели предметной области и почему он должен быть прозрачным и автоматическим. Предпочтительной программной моделью для достижения этого является POJO.

POJO

Аббревиатура POJO расшифровывается как Plain Old Java Objects (старые добрые объекты Java). Термин был введен в употребление Мартином Фаулером (Martin Fowler), Ребеккой Парсонс (Rebecca Parsons) и Джошем Маккензи (Josh Mackenzie) в 2000 г.

Около десяти лет назад разработчики заговорили о POJO – подходе, ведущем назад к основам, который, по сути, воскрешает JavaBeans, компонентную модель разработки пользовательских интерфейсов, и применяет ее к другому уровню системы. После нескольких ревизий спецификаций EJB и JPA появились легковесные сущности, которые правильно было бы называть *JavaBeans с возможностью сохранения*. Java-инженеры часто используют эти термины как синонимы для обозначения одного и того же базового подхода к проектированию.

Вам не следует слишком беспокоиться о терминах, используемых в книге; конечной целью является как можно более прозрачное добавление аспекта хранения в Java-классы. Практически каждый Java-класс можно сделать хранимым, если следовать некоторым простым советам. Давайте посмотрим, как это выглядит в коде.

3.2.3. Создание классов с возможностью сохранения

Главной задачей Hibernate является работа со сложными, детализированными предметными моделями. По этой причине мы работаем с POJO. В общем, использование детализированных объектов подразумевает большее количество классов, чем таблиц.

Старый добрый Java-класс с возможностью сохранения объявляет атрибуты, которые представляют состояние, и бизнес-методы, которые представляют поведение. Некоторые атрибуты представляют ассоциации с другими хранимыми классами.

POJO-реализация для сущности `User` предметной модели показана в следующем листинге. Давайте пройдемся по коду.

Листинг 3.1 ❖ POJO-реализация для сущности `User`

Файл:/model/src/main/java/org/jpwh/model/simple/User.java

```
public class User implements Serializable {

    protected String username;

    public User() {
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public BigDecimal calcShippingCosts(Address fromLocation) {
        // Пустая реализация бизнес-метода
        return null;
    }

    // ...
}
```

JPA не требует, чтобы хранимые классы реализовали интерфейс `java.io.Serializable`. Но когда экземпляры сохраняются в `HttpSession` или передаются по значению с помощью механизма удаленного вызова методов (Remote Method Invocation, RMI), сериализация необходима. В вашем приложении этот механизм может не использоваться, но класс будет сериализуемым без какой-либо дополнительной работы, и от этого не будет проблем (мы не станем объявлять классы сериализуемыми в каждом примере, подразумевая, что вы понимаете, когда это необходимо).

Класс может быть абстрактным и, если необходимо, расширять нехранимый класс или реализовать интерфейс. Он должен быть классом верхнего уровня, а не вложенным внутрь другого класса. Ни хранимые классы, ни их методы не могут быть финальными (требование спецификации JPA).

В отличие от спецификации JavaBeans, которая не требует наличия какого-либо специального конструктора, Hibernate (и JPA) требует, чтобы каждый хранимый класс имел конструктор без аргументов. С другой стороны, можно вообще обойтись без конструктора – Hibernate будет использовать Java-конструктор по умолчанию. Такой конструктор без аргументов вызывается для создания экземпляров при помощи механизма рефлексии в Java. Конструктор может не быть общедоступным, но он должен как минимум иметь область видимости уровня пакета, если для оптимизации производительности Hibernate должен будет использовать прокси-классы, сгенерированные во время выполнения. Рассмотрим также требования других спецификаций: стандарт EJB требует, чтобы у сеансовых компонентов (session beans) конструкторы были общедоступными, а спецификация JavaServer Faces (JSF) требует того же для управляемых компонентов (managed beans). Существуют и другие ситуации, когда может понадобиться общедоступный конструктор для создания «пустого» состояния – например, построение запроса по образцу.

Свойства POJO представляют атрибуты бизнес-сущностей – например, свойство `username` представляет имя пользователя (`User`). Свойства обычно реализованы как приватные или защищенные переменные-члены класса вместе с общедоступными или защищенными методами доступа – для каждого поля класса есть метод получения и метод установки значения. Они известны как методы *чтения* (getter) и *записи* (setter) соответственно. Объект POJO в листинге 3.1 имеет методы чтения и записи для свойства `username`.

Спецификация JavaBean определяет принципы именования методов доступа; это позволяет таким инструментам общего назначения, как Hibernate, с легкостью находить их и управлять значениями свойств. Имя метода чтения должно начинаться со слова `get`, за которым следует имя свойства (первая буква прописная); имя метода записи должно начинаться со слова `set` и аналогично заканчиваться именем свойства. В именах методов чтения логических свойств (типа `Boolean`) допускается вместо `get` использовать префикс `is`.

Hibernate не требует создания методов доступа. Вы выбираете, как должно сохраняться состояние экземпляра хранимого класса, а Hibernate будет либо обращаться к полям класса напрямую, либо станет вызывать методы доступа. Эти соображения не должны оказывать сильного влияния на дизайн классов. Вы можете сделать какие-то методы доступа приватными или защищенными либо вовсе избавиться от них – просто настройте Hibernate на доступ к полям класса для этих свойств.

Должны ли поля класса и методы доступа быть приватными, защищенными или видимыми на уровне пакета?

Как правило, желательно запрещать прямой доступ к внутреннему состоянию классов, поэтому поля классов не делают общедоступными. Объявляя поля класса или его методы приватными, вы фактически заявляете, что никто и никогда не должен иметь к ним доступа; только вы можете сделать это (или служба, подобная Hiber-

nate). Это категоричное заявление. Иногда у кого-нибудь могут возникнуть веские причины для доступа к вашему «приватному» содержимому – обычно, чтобы исправить одну из ваших ошибок, – и вы только разозлите остальных, если в экстренном случае им придется прибегать к механизму рефлексии. Вместо этого предпочтительнее предполагать, что инженер, пришедший после вас, которому потребуется доступ к вашему коду, знает, что делает.
Защищенная видимость в таком случае является более разумным выбором. Вы запрещаете прямой публичный доступ, показывая, что детали этого конкретного поля класса являются внутренними, но открываете доступ подклассам на случай необходимости. Вы доверяете инженеру – создателю подкласса. Ограничение видимости рамками пакета – не лучшее решение: вы заставляете кого-то создавать код в том же пакете для доступа к переменным-членам и методам, т. е. делать лишнюю работу без веской на то причины. Более важно, что эти рекомендации по выбору области видимости актуальны и в средах без политик безопасности и наличия объекта `SecurityManager` в среде выполнения. Если вам нужно сделать свой внутренний код приватным – делайте его приватным.

Тривиальные методы доступа широко используются, но мы обычно предпочитаем методы доступа в стиле JavaBeans, потому что они обеспечивают инкапсуляцию – вы можете поменять внутреннюю реализацию атрибута, не изменяя публичного интерфейса. Если вы настроите Hibernate на доступ к атрибутам через методы, вы абстрагируете внутреннюю структуру данных класса – переменные экземпляра – от схемы базы данных.

Например, если имя пользователя хранится в базе данных в одном столбце `NAME`, но в классе `User` есть поля `firstname` и `lastname`, вы можете определить в классе хранимое свойство `name`.

Листинг 3.2 ❖ Реализация класса `User` в виде POJO с логикой в методах доступа

```
public class User {

    protected String firstname;
    protected String lastname;

    public String getName() {
    return firstname + ' ' + lastname;
    }

    public void setName(String name) {
        StringTokenizer t = new StringTokenizer(name);
        firstname = t.nextToken();
        lastname = t.nextToken();
    }
}
```

Позже вы увидите, что для большинства подобных ситуаций лучше подходит преобразователь пользовательских типов в службе хранения. Он позволяет получить несколько вариантов на выбор.

Еще один аспект, на который стоит обратить внимание, – это *сравнение состояния объектов* (dirty checking). Hibernate автоматически определяет изменение со-

стояния для синхронизации с базой данных. Как правило, можно без проблем возвращать из метода чтения не тот экземпляр, который Hibernate передал в метод записи. Hibernate сравнит их значения, а не идентичность, чтобы выяснить, нужно ли обновлять сохраненное состояние атрибута. Следующий метод, к примеру, не станет причиной ненужных SQL-выражений `UPDATE`:

```
public String getFirstname() {    ← Это нормально
    return new String(firstname);
}
```

Но есть одно важное исключение – для коллекций проверяется идентичность! Для свойства, представляющего хранимую коллекцию, метод чтения должен возвращать ту же коллекцию, которую фреймворк Hibernate передал в метод записи. В противном случае Hibernate будет обновлять базу данных, даже если никакого обновления не требуется, для синхронизации состояния в памяти и в базе данных. Обычно вы должны стараться избегать подобного кода в методах потомков:

```
protected String[] names = new String[0];

public void setNames(List<String> names) {
    this.names = names.toArray(new String[names.size()]);
}

public List<String> getNames() {
    return Arrays.asList(names);    ← Не делайте этого, если Hibernate обращается к методам доступа!
}
```

Конечно, это не является проблемой, если Hibernate обращается к переменной-члену `names` непосредственно, в обход методов доступа.

Как Hibernate обрабатывает исключения, возбуждаемые методами доступа? Если метод доступа во время сохранения/извлечения экземпляра возбудит исключение `RuntimeException` (неконтролируемое), Hibernate откатит текущую транзакцию, и вы сможете обработать исключение в коде, вызвавшем Java Persistence API (или Hibernate). Если метод возбудит контролируемое исключение, Hibernate завернет его в `RuntimeException`.

Пример в листинге 3.1 также определяет метод расчета стоимости доставки товара конкретному пользователю (мы опустили его реализацию).

Далее мы сфокусируемся на отношениях между сущностями и ассоциациях между хранимыми классами.

3.2.4. Реализация ассоциаций в POJO

В этом разделе вы узнаете, как создавать различные виды отношений между объектами: *один к одному*, *многие к одному* и двунаправленные отношения. Мы рассмотрим вспомогательный код (scaffolding code), необходимый для создания этих ассоциаций, и как обеспечивать целостность этих отношений.

Для выражения ассоциаций между классами создаются свойства, и вы (как правило) вызываете методы доступа к ним для навигации от экземпляра к экземпля-

ру во время выполнения. Давайте рассмотрим ассоциации, определенные между хранимыми классами `Bid` и `Item`, показанные на рис. 3.4:

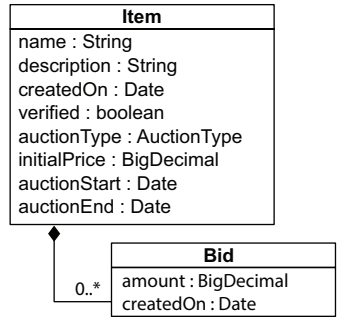


Рис. 3.4 ❖ Ассоциации между классами Bid и Item

Так же как на других UML-диаграммах, мы не опустили атрибутов, связанных с ассоциациями: `Item#bids` и `Bid#item`. Эти свойства и методы управления их значениями называются *вспомогательным кодом*. Так выглядит вспомогательный код для класса `Bid`:

Файл:/model/src/main/java/org/jpwh/model/simple/Bid.java

```
public class Bid {

    protected Item item;

    public Item getItem() {
        return item;
    }

    public void setItem(Item item) {
        this.item = item;
    }
}
```

Свойство `item` позволяет осуществить переход от предложения цены (`Bid`) к связанному товару (`Item`). Это – ассоциация вида *многие к одному*: пользователи могут делать несколько предложений цены для одного товара. А вот вспомогательный код класса `Item`:

Файл:/model/src/main/java/org/jpwh/model/simple/Item.java

```
public class Item {

    protected Set<Bid> bids = new HashSet<Bid>();

    public Set<Bid> getBids() {
        return bids;
    }
```

```
    public void setBids(Set<Bid> bids) {
        this.bids = bids;
    }
}
```

Ассоциация между двумя классами позволяет осуществлять *двунаправленную* (bidirectional) навигацию: связь *многие к одному* с этой точки зрения – то же, что и связь *один ко многим* (напомним, у одного товара может быть несколько предложений цены). Вспомогательный код для свойства `bids` использует тип интерфейса коллекции `java.util.Set`. JPA требует применения интерфейсов для полей-коллекций, и вы должны использовать `java.util.Set`, `java.util.List` или `java.util.Collection`, а не, к примеру, `HashSet`. Вообще, использование интерфейсов вместо конкретных реализаций является хорошей практикой, поэтому данное ограничение не должно вас беспокоить.

Выбрав `Set`, вы инициализируете поле класса новым объектом `HashSet`, потому что приложение не позволяет делать двух одинаковых предложений. Это хорошая практика, потому что предотвращает любое исключение типа `NullPointerException`, когда кто-нибудь обращается к свойству нового экземпляра `Item`, у которого отсутствуют предложения цены. Реализация JPA также обязана присвоить непустое значение любому отображаемому полю-коллекции – например, когда товар (`Item`) без предложений цены загружается из базы данных. (Необязательно использовать `HashSet` – выбор типа остается на усмотрение реализации. В Hibernate есть собственные реализации коллекций с дополнительными возможностями, такими как сравнение состояния объектов.)

Не лучше ли хранить предложения цены за товар в списке?

Обычно первое решение – запоминать тот порядок элементов, в котором их вводят пользователи, потому что позже вы, возможно, захотите показывать их в таком же порядке. Безусловно, в приложении для аукциона должен существовать порядок, в котором пользователю будут показываться предложения цены за товар: например, самая высокая цена показывается первой или самое последнее предложение – последним. Вы могли бы даже использовать `java.util.List` в вашем пользовательском интерфейсе для сортировки и отображения предложений цены за товар. Но это не означает, что порядок отображения должен сохраняться: целостность данных не зависит от порядка, в котором показываются предложения цен. Вам необходимо хранить значение цены, чтобы можно было найти наибольшую, и метку времени для каждого предложения, чтобы определить последнее. Если у вас есть сомнения, сделайте вашу систему гибкой и сортируйте данные при загрузке из хранилища данных (в запросе) и/или при показе пользователю (в Java-коде), а не при сохранении.

Так же как основные свойства, методы доступа к ассоциациям стоит делать общедоступными, только если они являются частью внешнего интерфейса хранимого класса и используются логикой приложения для создания связи между двумя экземплярами. Мы сейчас подробно рассмотрим этот вопрос, т. к. управление

связью между товаром (`Item`) и предложением цены (`Bid`) в Java гораздо сложнее, чем в базе данных SQL с декларативными ограничениями внешнего ключа. По нашему опыту, большинство инженеров незнакомо с этим затруднением, возникающим в графе объектов с двунаправленными ссылками (указателями). Давайте последовательно разберемся с этим вопросом.

Основной способ привязки предложения цены (`Bid`) к товару (`Item`) выглядит следующим образом:

```
anItem.getBids().add(aBid);
aBid.setItem(anItem);
```

Когда бы ни создавалась эта двунаправленная связь, необходимо выполнить два действия:

- добавить экземпляр `Bid` в коллекцию `bids` объекта `Item`;
- установить свойство `item` экземпляра `Bid`.

JPA не управляет ассоциациями. Если вы хотите управлять ассоциациями, то должны писать такой же код, какой создали бы без Hibernate. Если ассоциация двунаправленная, то вы должны помнить об обеих сторонах отношения. Если у вас когда-нибудь возникнут проблемы с пониманием поведения ассоциаций в JPA, задайте себе вопрос: «Что бы я сделал в отсутствие Hibernate?» Hibernate не изменяет обычную семантику Java.

Мы советуем создавать вспомогательные методы, объединяющие эти операции, которые обеспечивают повторное использование кода, помогают убедиться в корректности и гарантируют целостность данных (у предложения цены (`Bid`) должна быть ссылка на товар (`Item`)). Следующий листинг показывает такой вспомогательный метод для класса `Item`.

Листинг 3.3 ❖ Вспомогательный метод упрощает управление отношениями

Файл:/model/src/main/java/org/jpwh/model/simple/Item.java

```
public void addBid(Bid bid) {
    if (bid == null)                    ← Используйте защитное программирование
        throw new NullPointerException("Can't add null Bid");
    if (bid.getItem() != null)
        throw new IllegalStateException("Bid is already assigned to an Item");
    getBids().add(bid);
    bid.setItem(this);
}
```

Метод `addBid()` не только сокращает количество строк кода при работе с экземплярами `Item` и `Bid`, но и обеспечивает множественный характер ассоциации. Избегайте ошибок, отказываясь от выполнения одного из необходимых действий. По возможности всегда объединяйте операции над ассоциациями подобным образом. Если сравнить эту реализацию с реляционной моделью внешних ключей в базе данных SQL, легко можно увидеть, как модель, включающая граф объектов и указатели, может усложнить простую операцию: вместо декларативного ограничения потребуется написать процедурный код, гарантирующий целостность данных.

Поскольку желательно, чтобы `addBid()` оставался единственным методом, изменяющим предложения цен на товар, – возможно, как дополнение к методу `removeBid()`, – можно сделать метод `Item#setBids()` приватным или избавиться от него, настроив Hibernate на прямой доступ к полям класса для поддержки хранения. По той же причине для метода `Bid#setItem()` можно установить видимость на уровне пакета.

Метод чтения `Item#getBids()` по-прежнему возвращает изменяемую коллекцию, так что клиенты могут использовать ее для внесения изменений, которые не будут отражаться на противоположной стороне. Предложения цены, помещенные непосредственно в коллекцию, не будут ссылаться на товар: согласно ограничениям базы данных, это является противоречивым состоянием. Чтобы избежать этого, можно обернуть внутреннюю коллекцию перед возвратом с помощью `Collections.unmodifiableCollection(c)`и `Collections.unmodifiableSet(s)`. В этом случае клиент получит исключение при попытке изменить коллекцию; вы, таким образом, вынуждаете делать каждое изменение с помощью методов, управляющих отношениями, что обеспечивает целостность данных. Обратите внимание, что в этом случае необходимо настроить Hibernate на прямой доступ к полям класса, поскольку коллекция, возвращаемая методом чтения, будет отличаться от переданной в метод записи.

Альтернативой является использование неизменяемых экземпляров. Например, целостность данных можно обеспечить, требуя передачи аргумента типа `Item` в конструктор класса `Bid`, как показано в листинге 3.4.

Листинг 3.4 ❖ Обеспечение целостности отношений с помощью конструктора

Файл:/model/src/main/java/org/jpwh/model/simple/Bid.java

```
public class Bid {

    protected Item item;

    public Bid(Item item) {
        this.item = item;
        item.getBids().add(this);    ← Двунаправленный
    }

    public Item getItem() {
        return item;
    }
}
```

Этот конструктор присваивает значение полю `item` – его значение больше не должно меняться. Коллекция с «другой» стороны тоже обновляется для поддержания двустороннего отношения. Метод `Bid#setItem()` отсутствует, и, возможно, не следует также делать общедоступным метод `Item#setBids()`.

У такого подхода есть несколько проблем. Во-первых, Hibernate не сможет вызвать этот конструктор. Hibernate может использовать только конструктор без аргументов, и он должен иметь, по меньшей мере, видимость пакета. Кроме того, по-

скольку отсутствует метод `setItem()`, необходимо настроить Hibernate на прямой доступ к полю`item`. Это означает, что это поле не может иметь модификатор `final` и, следовательно, нельзя гарантировать неизменяемость класса.

В примерах из этой книги мы иногда будем создавать вспомогательные методы наподобие `Item#addBid()`, показанного выше, или дополнительные конструкторы для нужных значений. Только вам решать, сколько вспомогательных методов и слоев использовать для обертывания хранимых свойств ассоциаций и/или полей, но мы бы советовали быть последовательными и придерживаться одной стратегии для всех классов предметной модели. Для краткости мы не всегда будем показывать вспомогательные методы, специальные конструкторы и прочий дополнительный код в будущих примерах и рассчитываем, что вы сами добавите их согласно своим предпочтениям и требованиям.

Вы уже рассмотрели классы предметной модели, узнали, как представлять их атрибуты и отношения между ними. Далее мы повысим уровень абстрагирования, добавляя в реализацию модели метаданные и описывая такие свойства, как правила валидации и хранения.

3.3. Метаданные предметной модели

Метаданные – это данные о данных, поэтому метаданные предметной модели являются данными о ней. Например, когда вы используете механизм рефлексии Java для определения имен классов предметной модели или их атрибутов, вы обращаетесь к метаданным этой модели.

Инструментам ORM также требуются метаданные для описания отображений между классами и таблицами, свойствами и столбцами, ассоциациями и внешними ключами, типами Java и типами SQL и т. д. Эти метаданные объектно-реляционного отображения управляют преобразованиями между различными системами типов и представлениями отношений в объектно-ориентированных и SQL-системах. JPA обладает прикладным программным интерфейсом для работы с метаданными, который можно вызвать для получения информации о хранимых свойствах предметной модели, таких как имена хранимых сущностей и атрибутов. Ваша первоочередная задача как инженера – создавать и сопровождать эти метаданные.

Стандарт JPA определяет два варианта объявления метаданных: при помощи аннотаций в Java-коде или внешних файлов XML-дескрипторов. В Hibernate имеются некоторые собственные расширения, также доступные в виде аннотаций и/или XML-дескрипторов. Как правило, в качестве основного источника метаданных отображений мы выбираем либо аннотации, либо XML. После прочтения этого раздела у вас будет необходимая информация для принятия обоснованного решения в собственном проекте.

Мы также обсудим спецификацию *Bean Valildation* (JSR 303) и то, как она обеспечивает декларативную валидацию классов предметной модели (или любых других классов). Проект *Hibernate Validator* является эталонной реализацией этой

спецификации. Ныне в качестве основного механизма объявления метаданных большинство инженеров предпочитает использовать аннотации.

3.3.1. Определение метаданных с помощью аннотаций

Большим достоинством аннотаций является их близость к описываемым данным. Приведем пример:

Файл:/model/src/main/java/org/jpwh/model/simple/Item.java

```
import javax.persistence.Entity;

@Entity
public class Item {

}
```

Стандартные аннотации JPA можно найти в пакете `javax.persistence`. В данном примере аннотация `@javax.persistence.Entity` объявляет класс `Item` хранимой сущностью. При этом все его атрибуты автоматически становятся хранимыми, с применением стратегии по умолчанию. Это значит, что вы можете загружать и сохранять экземпляры `Item`, и все его свойства будут частью управляемого состояния.

(Читая предыдущую главу, вы, возможно, заметили отсутствие обязательной аннотации `@Id` и свойства-идентификатора. Если вы захотите использовать пример `Item`, добавьте свойство-идентификатор. Мы обсудим свойства-идентификаторы снова в следующей главе, в разделе 4.2.)

Аннотации типобезопасны, и метаданные JPA включаются в файлы скомпилированных классов. Затем, во время запуска приложения, Hibernate читает классы и метаданные, используя механизм рефлексии Java. Интегрированная среда разработки (IDE) может с легкостью проверять и подсвечивать аннотации – в конце концов, это обычные типы Java. При проведении рефакторинга постоянно приходится переименовывать, удалять или перемещать классы и свойства. Большинство текстовых редакторов и инструментов для разработки не может проводить рефакторинг значений атрибутов и элементов XML, но аннотации являются частью языка Java и участвуют во всех операциях рефакторинга.

Мой класс теперь зависит от JPA?

Да, но это лишь зависимость времени компиляции. Вам потребуется обеспечить доступность библиотек JPA во время компиляции исходного кода классов вашей предметной модели. Наличие Java Persistence API не обязательно для создания экземпляров класса – например, в клиентском настольном приложении, не использующем JPA. Пакеты потребуются, только если вы обращаетесь к аннотациям через механизм рефлексии во время выполнения (так работает Hibernate, когда читает ваши метаданные).

Когда стандартизированных аннотаций Java Persistence недостаточно, реализация JPA может предоставлять дополнительные аннотации.

Использование расширений реализации JPA

Даже если большую часть прикладной модели можно отобразить с помощью JPA-совместимых аннотаций из пакета `javax.persistence`, в какой-то момент вам придется использовать расширения реализации. Например, некоторые настройки производительности, которые можно ожидать от высококачественного ПО, доступны только в качестве аннотаций Hibernate. Таким способом реализации JPA соревнуются между собой; так что вам не удастся избежать использования аннотаций из других пакетов – это одна из причин, почему рекомендуется выбирать Hibernate.

Следующее определение сущности `Item` демонстрирует возможность отображения, доступную только в Hibernate:

```
import javax.persistence.Entity;

@Entity
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
)
public class Item {

}
```

Мы предпочитаем присоединять полное имя пакета `org.hibernate.annotations` к аннотациям Hibernate. Считайте это хорошей практикой, т. к. вы легко сможете увидеть, какие метаданные класса относятся к спецификации JPA, а какие – к конкретной ее реализации. Также вы можете просто выполнить поиск по строке «org.hibernate.annotations» в исходном коде и получить список всех нестандартных аннотаций, используемых в приложении.

В случае смены реализации JPA вам нужно будет лишь заменить аннотации, относящиеся к конкретной реализации, аналогичными средствами, доступными во всех продвинутых реализациях JPA. Конечно, мы надеемся, что вам никогда не придется этого делать – и это нечасто встречается на практике, – просто знайте, что такое возможно.

Аннотации классов описывают метаданные, применимые лишь к этому конкретному классу. Часто требуются метаданные более высокого уровня – для всего пакета или даже всего приложения.

Метаданные и глобальные аннотации

Аннотация `@Entity` отображает конкретный класс. Но в JPA и Hibernate также имеются аннотации для глобальных метаданных. Например, аннотация `@NamedQuery` имеет глобальную область видимости; она применяется не к конкретному классу. Куда ее поместить?

Несмотря на то что такую глобальную аннотацию можно поместить в файл с исходным кодом класса (любого класса, в начало файла), мы предпочитаем хранить глобальные метаданные в отдельном файле. Хорошим выбором являются аннотации уровня пакета; они хранятся в файле с именем *package-info.java* в каталоге

конкретного пакета. Пример объявления глобальных именованных запросов показан в листинге 3.5.

Листинг 3.5 ❖ Глобальные метаданные в файле *package-info.java*

Файл:/model/src/main/java/org/jpwh/model/querying/package-info.java

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByName",
        query = "select i from Item i order by i.name asc"
    )
    ,
    @org.hibernate.annotations.NamedQuery(
        name = "findItemBuyNowPriceGreaterThan",
        query = "select i from Item i where i.buyNowPrice > :price",
        timeout = 60,                          ← Секунды!
        comment = "Custom SQL comment"
    )
})

package org.jpwh.model.querying;
```

Если вы не использовали аннотации уровня пакета прежде, вас, возможно, удивит синтаксис с объявлениями пакета и импорта внизу.

Существует причина, почему в предыдущем примере были только аннотации Hibernate и никаких аннотаций JPA. Мы не стали использовать стандартную аннотацию JPA `@org.javax.persistence.NamedQuery`, а выбрали альтернативную аннотацию Hibernate. Аннотации JPA не поддерживают видимость на уровне пакета по неизвестной нам причине. По сути, JPA не позволяет помещать аннотации в файл *package-info.java*. Оригинальные аннотации Hibernate предоставляют ту же самую, а иногда и большую функциональность, так что это не должно стать большой проблемой. Если вы не хотите использовать аннотации Hibernate, вам придется либо включить аннотации JPA в начало одного из классов (с другой стороны, можно создать пустой класс `MyNamedQueries` и сделать его частью предметной модели), либо использовать файл XML, как будет показано ниже в этом разделе.

Аннотации будут нашим главным инструментом определения метаданных ORM на протяжении всей книги, и вы многое узнаете о них.

3.3.2. Применение правил валидации компонентов

Большинство приложений использует огромное количество проверок целостности данных. Вы уже знаете, что бывает, если нарушить хотя бы одно из простейших ограничений целостности данных: вы получите `NullPointerException` там, где ожидаете получить значение. В других примерах используются свойства строкового типа, которые не должны быть пустыми (помните, что пустая строка не `null`), строки, которые должны соответствовать определенному шаблону регулярного выражения, а также числа и даты, которые должны находиться в определенном диапазоне.

Эти бизнес-правила влияют на каждый уровень приложения. Пользовательский интерфейс должен отображать подробные и локализованные сообщения об ошибках. Уровни бизнес-логики и хранения должны проверять приходящие от клиентов значения, прежде чем передавать их в хранилище. База данных SQL должна делать окончательную проверку, чтобы гарантировать целостность хранимой информации.

Суть идеи, лежащей в основе Bean Validation, – в том, что определять такие правила, как «Это поле не может быть null» или «Это число должно находиться в заданном диапазоне», гораздо проще, чем постоянно писать подверженные ошибкам процедуры проверок с использованием условных операторов. Более того, задание таких правил для центрального компонента приложения – реализации предметной модели – позволяет осуществлять проверку целостности на каждом уровне системы. Эти правила доступны уровням представления и хранения. Если вы посмотрите, как ограничения целостности данных влияют не только на код приложения, но и на схему базы данных SQL, являющуюся набором правил поддержания целостности, вы сможете представить ограничения Bean Validation как дополнительные метаданные ORM.

Посмотрите на расширенную версию класса `Item` предметной модели.

Листинг 3.6 ❖ Применение правил валидации к свойствам сущности `Item`

Файл:/model/src/main/java/org/jpwh/model/simple/Item.java

```
import javax.validation.constraints.Future;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
public class Item {

    @NotNull
    @Size(
        min = 2,
        max = 255,
        message = "Name is required, maximum 255 characters."
    )
    protected String name;
    @Future
    protected Date auctionEnd;
}
```

Здесь добавлены еще два атрибута: наименование товара (`name`) и дата окончания аукциона (`auctionEnd`). Оба являются типичными кандидатами для дополнительных ограничений – важно гарантировать присутствие осмысленного наименования (наименование из одной буквы бессмысленно), но не слишком длинного: база данных SQL работает наиболее эффективно со строками произвольной длины до 255 символов, и пользовательский интерфейс также имеет ограничения на пространство, занимаемое видимой надписью. Время окончания аукциона, очевидно, должно быть в будущем. Если вы не подготовите сообщения об ошибке,

будет показано сообщение по умолчанию. Сообщения могут представлять собой ключи к внешним файлам свойств, используемым для интернационализации.

Механизм валидации будет обращаться напрямую к полям класса, если их пометить аннотациями. Если вы предпочитаете методы доступа, помечать аннотацией проверки ограничений нужно метод чтения, а не записи. Ограничения являются также частью API класса и помещаются в Javadoc, что делает реализацию предметной модели более понятной. Обратите внимание, что механизм валидации имеет свои настройки, не зависящие от настроек реализации JPA, т. е. Hibernate Validator может вызывать методы доступа, тогда как Hibernate ORM может обращаться напрямую к полям класса.

Bean Validation не ограничивается встроенными аннотациями; вы можете создавать собственные ограничения и аннотации. Пользовательские ограничения позволяют даже применять аннотации на уровне класса и проверять несколько атрибутов экземпляра класса одновременно. Следующий тестовый код демонстрирует, как вручную проверить целостность экземпляра `Item`:

Листинг 3.7 ❖ Проверка экземпляра `Item` на предмет нарушения ограничений

Файл:/examples/src/test/java/org/jpwh/test/simple/ModelOperations.java

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Item item = new Item();
item.setName("Some Item");
item.setAuctionEnd(new Date());

Set<ConstraintViolation<Item>> violations = validator.validate(item);

assertEquals(1, violations.size());   <-- Одна ошибка валидации: дата аукциона не находится в будущем

ConstraintViolation<Item> violation = violations.iterator().next();
String failedPropertyName =
        violation.getPropertyPath().iterator().next().getName();
assertEquals(failedPropertyName, "auctionEnd");

if (Locale.getDefault().getLanguage().equals("en"))
    assertEquals(violation.getMessage(), "must be in the future");
```

Мы не будем подробно объяснять этот код, а предоставим его исследование читателю. Вам редко придется писать проверки такого рода: большую часть времени эта работа будет выполняться вашим пользовательским интерфейсом и фреймворком хранения. Поэтому важно обратить внимание на интеграцию с Bean Validation при выборе фреймворка пользовательского интерфейса. JSF версии 2.0 и выше, к примеру, автоматически интегрируется с Bean Validation.

Hibernate, как это требуется от любой реализации JPA, автоматически интегрируется с Hibernate Validator, если библиотеки доступны в пути поиска классов, и предлагает следующие возможности:

- она требует вручную осуществлять валидацию экземпляров перед передачей их на сохранение;

- Hibernate распознает ограничения хранимых классов предметной модели и вызывает валидацию перед операциями вставки и обновления базы данных. В случае выявления нарушений Hibernate вызывает исключение `ConstraintViolationException` с информацией об ошибке в коде, который вызвал операцию сохранения;
- инструментарий Hibernate, автоматически генерирующий SQL-схемы, понимает многие ограничения и создает аналогичные ограничения SQL в DDL-определениях данных. Например, аннотация `@NotNull` транслируется в SQL-ограничение `NOT NULL`, а правило `@Size(n)` задает количество символов в столбце типа `VARCHAR(n)`.

Вы можете управлять этим поведением Hibernate, используя элемент `<validation-mode>` в конфигурационном файле *persistence.xml*. По умолчанию используется режим `AUTO`, следовательно, Hibernate будет осуществлять валидацию, только если найдет реализацию Bean Validation (например, Hibernate Validator) в пути поиска классов запущенного приложения. В режиме `CALLBACK` валидация выполняется всегда, и вы получите ошибку развертывания (deployment error), если забудете скомпоновать свое приложение с реализацией Bean Validation. Режим `NONE` отключает автоматическую валидацию реализацией JPA.

Вы еще увидите аннотации Bean Validation далее в этой книге; вы также найдете их в комплекте примеров. Сейчас мы могли бы написать гораздо больше о Hibernate Validator, но мы лишь повторили бы ту информацию, что уже содержится в отличном справочном руководстве проекта. Прочтите ее, чтобы узнать о таких особенностях, как группы валидации или прикладной интерфейс обнаружения ограничений в метаданных.

Стандарты Java Persistence и Bean Validation широко используют аннотации. Экспертные группы были хорошо осведомлены о преимуществах XML-дескрипторов развертывания в определенных условиях, особенно для определения метаданных, меняющихся при каждом развертывании.

3.3.3. Метаданные во внешних XML-файлах

Вы можете заменить или переопределить любую аннотацию JPA при помощи элемента XML-дескриптора. Другими словами, вы не обязаны использовать аннотации, если не хотите этого или когда размещение метаданных отдельно от исходного кода по каким-то причинам выгодно для вашего проекта.

XML-метаданные и JPA

В листинге 3.8 показан XML-дескриптор JPA для конкретной единицы хранения.

Листинг 3.8 ❖ XML-дескриптор JPA с метаданными отображения единицы хранения

```
<entity-mappings
        version="2.1"
        xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
            http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">
    <persistence-unit-metadata>
        <xml-mapping-metadata-complete/>
        <persistence-unit-defaults>
            <delimited-identifiers/>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    <entity class="org.jpwh.model.simple.Item" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value strategy="AUTO"/>
            </id>
            <basic name="name"/>
            <basic name="auctionEnd">
                <temporal>TIMESTAMP</temporal>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

Сначала глобальные метаданные

Игнорировать все аннотации и любые метаданные отображения в XML-файлах

Некоторые настройки по умолчанию

Экранировать все SQL-имена, включая столбцы и таблицы: например, если ваши SQL-имена являются ключевыми словами (такими как «USER»)

Реализация JPA начнет автоматически использовать этот дескриптор, если поместить его в файл *META-INF/orm.xml*, находящийся в пути поиска классов единицы хранения. Если вы предпочитаете использовать другое имя или несколько файлов, придется поменять конфигурацию единицы хранения в файле *META-INF/persistence.xml*:

Файл:/model/src/main/resources/META-INF/persistence.xml

```
<persistence-unit name="SimpleXMLCompletePU">
    ...
    <mapping-file>simple/Mappings.xml</mapping-file>
    <mapping-file>simple/Queries.xml</mapping-file>
    ...
</persistence-unit>
```

При наличии элемента `<xml-mapping-metadata-complete>` реализация JPA будет игнорировать все аннотации в классах предметной модели в этой единице хранения и полагаться лишь на отображения, определенные в XML-дескрипторе(ах). Также можно (в данном случае это лишнее) активировать эту настройку на уровне сущности, используя `<metadata-complete="true"/>`. При активации реализация JPA предполагает, что вы отобразили все атрибуты сущности в XML, и ей следует игнорировать все аннотации в этой конкретной сущности.

Если вместо этого требуется не игнорировать, а переопределить метаданные, указанные в аннотации, – не помечайте XML-дескриптор как «завершенный» и укажите класс и свойство для переопределения:

```
<entity class="org.jpwh.model.simple.Item">
    <attributes>
        <basic name="name">    ← Переопределить имя столбца SQL
            <column name="ITEM_NAME"/>
        </basic>
    </attributes>
</entity>
```

Здесь свойство `name` отображается в столбец `ITEM_NAME`; по умолчанию свойство отображалось бы в столбец `NAME`. Теперь Hibernate будет игнорировать все аннотации из пакетов `javax.persistence.annotation` и `org.hibernate.annotations`, которыми отмечено поле `name` класса `Item`. Но Hibernate не станет игнорировать аннотации Bean Validation и продолжит использовать их для автоматической валидации и создания схемы! Все остальные аннотации для класса `Item` также будут распознаваться. Обратите внимание, что в этом отображении не определена стратегия доступа, поэтому, в зависимости от расположения аннотации `@Id` в классе `Item`, будет использоваться либо прямое обращение к полю, либо методы доступа (мы обсудим эту подробность в следующей главе).

В этой книге мы не станем много говорить о XML-дескрипторах JPA. Синтаксис этих документов является зеркальным отражением синтаксиса JPA-аннотаций, так что у вас не должно возникнуть проблем с их созданием. Мы заострим внимание на важном вопросе – стратегиях отображения. Синтаксис, используемый для записи метаданных, является вторичным.

К сожалению, как и большинство схем в JavaEE, JPA-схема *orm_2_0.xsd* не допускает расширения реализациями. Вы не сможете использовать элементы и атрибуты из других пространств имен в XML-документах отображений JPA. Следовательно, использование расширений и оригинальных особенностей Hibernate требует использования иного синтаксиса XML.

XML-файлы отображений Hibernate

Оригинальный формат XML-файлов отображений Hibernate был первоначальным вариантом метаданных до того, как в JDK 5 появились аннотации. По соглашению к именам этих файлов добавляется расширение *.hbm.xml*. Листинг 3.9 демонстрирует основной XML-документ отображения Hibernate:

Листинг 3.9 ❖ Документ определения метаданных с оригинальным XML-синтаксисом Hibernate

Файл:/model/src/main/resources/simple/Native.hbm.xml

```
<?xml version="1.0"?>
<hibernate-mapping
        xmlns="http://www.hibernate.org/xsd/orm/hbm"
        package="org.jpwh.model.simple"
        default-access="field">    ← ❶ Объявление метаданных

    <class name="Item">    ← Отображение класса сущности
        <id name="id">
```

```
            <generator class="native"/>
        </id>
        <property name="name"/>
        <property name="auctionEnd" type="timestamp"/>
    </class>

    <query name="findItemsHibernate">select i from Item i</query>

    <database-object>
        <create>create index ITEM_NAME_IDX on ITEM(NAME)</create>
        <drop>drop index if exists
     ITEM_NAME_IDX</drop>
    </database-object>

</hibernate-mapping>
```

Запросы, вынесенные в отдельный файл

Вспомогательные определения DDL

❶ Метаданные объявляются в корневом элементе `<hibernate-mapping>`. Такие атрибуты, как имя пакета и доступ по умолчанию, применяются ко всем отображениям в этом файле. Вы можете добавить столько отображений классов сущностей, сколько захотите.

Обратите внимание, что в этом файле объявляется пространство имен XML по умолчанию для всех элементов; это новая возможность в Hibernate 5. Если у вас уже имеются файлы отображений для Hibernate 4 или старше, содержащие определения типа документа XML, можете продолжать их использовать.

Несмотря на возможность объявления отображений для нескольких классов в одном файле, включив несколько элементов `<class>`, большое количество старых проектов Hibernate использует один файл отображения для каждого хранимого класса. По соглашению файл должен иметь то же имя и пакет, что и класс: например, *my/model/Item.hbm.xml* – для класса `my.model.Item`.

Отображение класса в XML-документе Hibernate является «завершенным», т. е. любые другие метаданные отображения для этого класса в аннотациях или XML-файлах JPA вызовут ошибку «duplicate mapping» (дублирование отображения) при запуске. Если вы описываете отображение класса в XML-файле Hibernate, это описание должно включать все детали отображения. Невозможно переопределить отдельные поля или расширить существующее отображение. Кроме того, в XML-файле Hibernate нужно перечислить и отобразить все хранимые свойства класса сущности. Если свойство не имеет отображения, Hibernate считает его состояние временным. Сравните это с отображениями в JPA, где одна лишь аннотация `@Entity` делает все свойства класса хранимыми.

Оригинальные XML-файлы Hibernate больше не являются предпочтительным способом объявления основных метаданных ORM. Большинство инженеров предпочитает аннотации. Оригинальные XML-файлы метаданных в основном используются для доступа к особенностям Hibernate, которые недоступны в виде аннотаций или которые легче сопровождать в XML-файлах (например, если это метаданные конфигурации, зависящие от условий развертывания). В XML-файле отображений Hibernate вообще могут *отсутствовать любые* элементы `<class>` . В таком случае все метаданные из этого файла, такие как строки запросов (или

даже запросы на чистом SQL), вынесенные в отдельный файл, определения пользовательских типов, вспомогательные выражения DDL для конкретных СУБД, динамические фильтры контекста хранения и т. д., могут быть глобальными для единицы хранения.

Когда позже мы будем обсуждать такие продвинутые возможности Hibernate, мы покажем, как объявлять все это в XML-файлах. Как уже упоминалось, вы должны сосредоточиться на понимании сущности стратегий отображения, и для их демонстрации большинство наших примеров будет использовать аннотации JPA и Hibernate.

Все рассматривавшиеся до сих пор подходы предполагали, что на момент разработки (развертывания) известны все метаданные ORM. Предположим, что какая-то часть информации остается неизвестной до запуска приложения. Можно ли программно манипулировать метаданными отображений во время выполнения? Также мы упоминали программный интерфейс метаданных JPA для доступа к подробным деталям единицы хранения. Как это работает и когда это полезно?

3.3.4. Доступ к метаданным во время выполнения

Спецификация JPA определяет программные интерфейсы для доступа к метамодели хранимых классов. Существуют два типа таких интерфейсов. Первый, более динамичный по природе, похож на простую рефлексию в Java. Второй представляет собой статическую метамодель, которая обычно создается процессором аннотаций Java 6. В обоих случаях доступ возможен только на чтение; невозможно менять метаданные во время выполнения.

Hibernate также предлагает оригинальный API метамодели, поддерживающий доступ как на чтение, так и на запись, а также доступ к большему количеству деталей ORM. Мы не будем рассматривать этот API (расположенный в `org.hibernate.cfg.Configuration`), поскольку на момент создания книги он уже был устаревшим, и не было API, его заменяющего. За последней информацией об этой возможности обращайтесь к документации Hibernate.

Динамический интерфейс Metamodel API в Java Persistence

Иногда – например, когда требуется написать собственный код валидации или обобщенный код пользовательского интерфейса, – желательно иметь программный доступ к хранимым атрибутам сущности. Так, порой желательно знать, какие хранимые классы и атрибуты имеются в предметной модели. Код в листинге 3.10 показывает, как читать метаданные с помощью интерфейсов Java Persistence.

Листинг 3.10 ❖ Получение информации о типе сущности с помощью Metamodel API

Файл:/examples/src/test/java/org/jpwh/test/simple/AccessJPAMetamodel.java

```
Metamodel mm = entityManagerFactory.getMetamodel();

Set<ManagedType<?>> managedTypes = mm.getManagedTypes();
assertEquals(managedTypes.size(), 1);
```

```
ManagedType itemType = managedTypes.iterator().next();
assertEquals(
    itemType.getPersistenceType(),
    Type.PersistenceType.ENTITY
);
```

Экземпляр `Metamodel` можно получить либо из объекта `EntityManagerFactory`, который обычно имеется в приложении в одном экземпляре на каждый источник данных, либо, если так удобнее, вызывая `EntityManager#getMetamodel()`. Множество управляемых типов содержит информацию обо всех хранимых сущностях и встроенных классах (которые мы рассмотрим в следующей главе). В этом примере есть только одна сущность `Item`. Вот как можно копнуть глубже и узнать больше о каждом атрибуте.

Листинг 3.11 ❖ Получение информации об атрибутах сущности с помощью Metamodel API

Файл:/examples/src/test/java/org/jpwh/test/simple/AccessJPAMetamodel.java

```
SingularAttribute nameAttribute =
    itemType.getSingularAttribute("name");    ← ❶ Атрибут сущности
assertEquals(
    nameAttribute.getJavaType(),
    String.class
);
assertEquals(
    nameAttribute.getPersistentAttributeType(),
    Attribute.PersistentAttributeType.BASIC
);
assertFalse(
    nameAttribute.isOptional()    ← Not null
);

SingularAttribute auctionEndAttribute =
    itemType.getSingularAttribute("auctionEnd");    ← ❷ Атрибут сущности
assertEquals(
    auctionEndAttribute.getJavaType(),
    Date.class
);
assertFalse(
    auctionEndAttribute.isCollection()
);
assertFalse(
    auctionEndAttribute.isAssociation()
);
```

Получение атрибутов сущности происходит с использованием строк `name` (❶) и `auctionEnd`(❷). Очевидно, это небезопасно с точки зрения типизации, и, если поменяются названия атрибутов, этот код перестанет работать. Строки не включаются автоматически в операции рефакторинга, выполняемые IDE.

JPA также предлагает статическую типобезопасную метамодель.

Статическая метамодель

Java (по крайней мере, до версии 8) не имеет полноценной поддержки свойств. Невозможно получить доступ к полям класса или методам доступа компонента типобезопасным способом – только по их именам с использованием строк. Особенно это неудобно при использовании запросов на основе критериев JPA, безопасной с точки зрения типизации альтернативы строковым языкам запросов. Вот пример:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Item> query =
    cb.createQuery(Item.class);     ← Эквивалент этого запроса: «select i from Item i»
Root<Item> fromItem = query.from(Item.class);
query.select(fromItem);

List<Item> items =
    entityManager.createQuery(query)
        .getResultList();
assertEquals(items.size(), 2);
```

Данный запрос возвращает все товары из базы данных; сейчас их два. Если вы хотите ограничить результат и вернуть только товары с определенными названиями, придется использовать выражение `like`, сравнивающее атрибут `name` каждого товара с шаблоном, заданным через параметр.

```
Path<String> namePath = fromItem.get("name");     ← «where i.name like :pattern»
query.where(
    cb.like(
        namePath,     ← Для оператора like() нужен экземпляр Path<String>!
        cb.parameter(String.class, "pattern")
    )
);

items =
    entityManager.createQuery(query)
        .setParameter("pattern", "%some item%") ← Произвольные последовательности символов!
        .getResultList();
assertEquals(items.size(), 1);
assertEquals(items.iterator().next().getName(), "This is some item");
```

Обратите внимание, что для поиска `namePath` требуется строка `name`. В этом самом месте нарушается безопасность типов в запросе на основе критериев. Вы можете переименовать класс сущности `Item` в процессе рефакторинга, и запрос будет по-прежнему работать. Но стоит только изменить поле `Item#name`, как потребуется вносить исправления вручную. К счастью, эта проблема легко обнаруживается с помощью тестов.

Лучшим решением, безопасным для рефакторинга и выявления несоответствий во время компиляции, а не выполнения, является типобезопасная статическая метамодель:

```
query.where(
    cb.like(
        fromItem.get(Item_.name),    ← Статическая метамодель Item!
        cb.parameter(String.class, "pattern")
    )
);
```

`Item_` – это особый класс (обратите внимание на подчеркивание). Он является классом метаданных и перечисляет все атрибуты класса сущности `Item`:

```
@javax.persistence.metamodel.StaticMetamodel(Item.class)
public abstract class Item_ {

    public static volatile SingularAttribute<Item, Long> id;
    public static volatile SingularAttribute<Item, String> name;
    public static volatile SingularAttribute<Item, Date> auctionEnd;
}
```

Этот класс можно написать вручную или, как и задумывали проектировщики, автоматически сгенерировать при помощи *утилиты обработки аннотаций* (annotation processing tool – apt) компилятора Java. *Hibernate JPA2 Metamodel Generator* (отдельный проект в семействе Hibernate) использует это расширение. Его единственной целью является создание статических классов метамодели из управляемых хранимых классов. Вы можете скачать его JAR-файл и интегрировать в свою IDE (или в инструмент сборки Maven, как сделано в примерах к этой книге). Он будет автоматически вызываться при каждой компиляции (или изменении, в зависимости от IDE) класса сущности `Item`, генерируя соответствующий класс метаданных `Item`.

Что такое утилита обработки аннотаций (apt)?

Java включает утилиту командной строки `apt` (annotation processing tool), которая находит и обрабатывает аннотации в исходном коде с помощью соответствующих процессоров. Для обработки аннотаций в программе процессор аннотаций использует механизм рефлексии (JSR 175). Программный интерфейс `apt` позволяет получить время сборки, имя исходного файла и представление, доступное только для чтения, для моделирования системы типов в Java. Процессоры аннотаций могут сначала создать новый исходный код и файлы, которые `apt` затем скомпилирует вместе с остальным исходным кодом.

В предыдущих разделах вы познакомились с некоторыми конструкциями отображений, но мы еще не представили более изощренных способов отображения классов и свойств. Сейчас вы должны определить, какую стратегию определения метаданных отображения использовать в своем проекте, – мы рекомендуем аннотации, а XML применять только в случае необходимости, – а затем, в следующей главе, узнать еще больше об отображениях классов и свойств.

3.4. Резюме

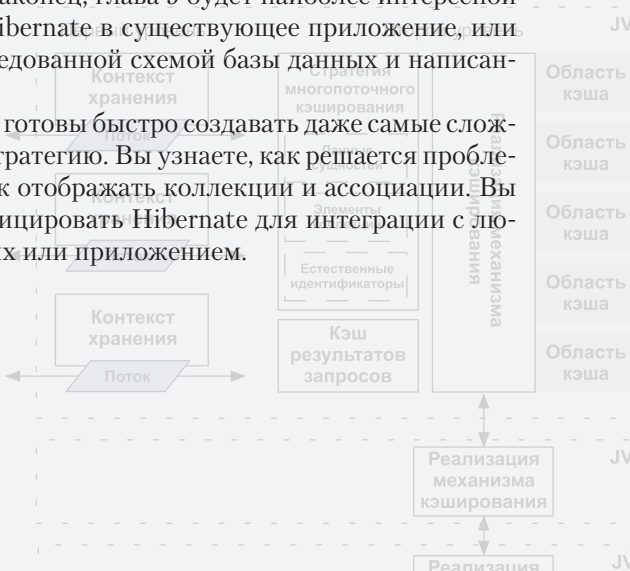
- Вы реализовали хранимые классы, свободные от сквозных задач, таких как журналирование, авторизация и разграничение транзакций; ваши хранимые классы зависят от JPA только во время компиляции. Даже аспекты, связанные с хранением, не должны проникать в реализацию предметной модели.
- Прозрачность работы механизма хранения необходима, если вы хотите исполнять и тестировать бизнес-объекты независимо и легко.
- Вы познакомились с общепризнанными практическими приемами и требованиями программной модели POJO и сущностей JPA, и узнали, что у них общего со старой спецификацией JavaBean.
- Вы уже готовы к определению более сложных отображений, возможно, используя комбинацию аннотаций JDK и/или XML-файлов отображения JPA/Hibernate.

Часть II

СТРАТЕГИИ ОТОБРАЖЕНИЯ

Эта часть целиком посвящена настоящему ORM – от классов и полей до таблиц и столбцов. Глава 4 начинается с рассмотрения обычных отображений полей и классов и объясняет, как отображать хорошо детализированные Java-модели предметной области. Далее, в главе 5, вы узнаете, как отображать простые поля и встраиваемые компоненты, а также как управлять отображением типов между Java и SQL. В главе 6 вы научитесь отображать в базу данных иерархии наследования сущностей, используя четыре основные стратегии отображения наследования; вы также познакомитесь с отображением полиморфных ассоциаций. Глава 7 целиком посвящена отображению коллекций и ассоциаций между сущностями – вы узнаете, как отображаются хранимые коллекции, коллекции базовых и встраиваемых типов, а также простые ассоциации *многие к одному* и *один ко многим* между сущностями. В главе 8 вы углубитесь в изучение продвинутых отображений ассоциаций сущностей – вы узнаете об отображении ассоциаций *один к одному*, вариантах отображения *один ко многим*, ассоциациях *многие ко многим* и тернарных отношения между сущностями. Наконец, глава 9 будет наиболее интересной для тех, кому необходимо внедрить Hibernate в существующее приложение, или если вам приходится работать с унаследованной схемой базы данных и написанным вручную SQL.

Прочитав эту часть книги, вы будете готовы быстро создавать даже самые сложные отображения, используя верную стратегию. Вы узнаете, как решается проблема с отображением наследования и как отображать коллекции и ассоциации. Вы также научитесь настраивать и модифицировать Hibernate для интеграции с любой существующей схемой базы данных или приложением.



Глава 4

Отображение хранимых классов

В этой главе:
- понятие сущности и типа-значения (value type);
- отображение классов сущностей с идентичностью;
- настройка отображений на уровне сущности.

В этой главе представлены некоторые фундаментальные свойства отображений и объясняется, как отображать классы сущностей в таблицы SQL. Мы покажем и обсудим, как обращаться с идентичностью в базе данных и первичными ключами, как использовать множество других настроек загрузки и сохранения экземпляров классов предметной модели в Hibernate. Все примеры отображений используют JPA-аннотации. Однако, прежде чем начать, мы определим коренное различие между сущностями и типами-значениями и объясним, как подходить к объектно-реляционному отображению предметной модели.

Главные нововведения в JPA2

С помощью элемента `<delimited-identifiers>` в конфигурационном файле *persistence.xml* можно включить глобальное экранирование всех имен в сгенерированных SQL-выражениях.

4.1. Понятие сущностей и типов-значений

Рассматривая модель предметной области, можно заметить различие между классами: те, что представляют бизнес-объекты (термин *объект* употреблен здесь в своем обычном смысле), кажутся более важными. Например, классы `Item`, `Category` и `User` – это сущности реального мира, которые вы пытаетесь представить (обратитесь к рис. 3.3, чтобы увидеть примерную модель предметной области). Другие типы в предметной модели, такие как `Address`, `String` и `Integer`, кажутся менее важными. В этом разделе мы посмотрим, что значит использовать хорошо

детализированную модель предметной области, делая различия между сущностями и типами-значениями.

4.1.1. Хорошо детализированные модели предметной области

Основной целью Hibernate является поддержка хорошо детализированных и насыщенных предметных моделей. Это одна из причин, по которой мы используем POJO. Проще говоря, *хорошо детализированный* означает больше классов, чем таблиц.

Например, в предметной модели у пользователя может быть домашний адрес. В базе данных может быть одна таблица `USER` со столбцами `HOME_STREET`, `HOME_CITY` и `HOME_ZIPCODE`. (Помните проблему SQL-типов, которую мы обсуждали в разделе 1.2.1?)

В предметной модели можно было бы применить тот же подход и представить адрес в виде трех строковых свойств класса `User`. Но гораздо лучше смоделировать это в виде класса `Address`, чтобы у пользователя было свойство `homeAddress`. Такая модель предметной области обладает лучшей связностью и большими возможностями для повторного использования кода, при этом она гораздо понятнее, чем SQL с негибкой системой типов.

JPA подчеркивает полезность хорошо детализированных классов для реализации типобезопасности и поведения. Например, многие моделируют адрес электронной почты как строковое свойство в классе `User`. Более сложным подходом было бы определение класса `EmailAddress`, добавляющего высокоуровневую семантику и поведение, – он мог бы предоставлять метод `prepareMail()` (у него не должно быть метода `sendMail()`, потому что класс предметной модели не должен зависеть от почтовой подсистемы).

Эта проблема детализации приводит нас к различию, имеющему чрезвычайную важность в ORM. В Java все классы равны: у каждого экземпляра свои идентичность и жизненный цикл. Когда вы добавляете поддержку долговременного хранения, может случиться так, что некоторые объекты не имеют собственной идентичности и жизненного цикла – они зависят от других объектов. Давайте рассмотрим пример.

4.1.2. Определение сущностей приложения

В одном доме живут два человека, и оба регистрируют учетные записи в приложении CaveatEmptor. Назовем их Джон (John) и Джейн (Jane).

Каждая учетная запись представлена экземпляром `User`. Экземпляры `User` должны загружаться, сохраняться и удаляться независимо, соответственно, `User` является классом сущности, а не типом-значением. Находить классы сущностей легко.

У класса `User` есть поле `homeAddress` – ассоциация с классом `Address`. Ссылаются ли оба экземпляра `User` во время выполнения на один и тот же экземпляр `Address`, или каждый экземпляр `User` ссылается на свой собственный экземпляр `Address`? Имеет ли значение, что Джейн и Джон живут в одном доме?

На рис. 4.1 показано, как два экземпляра `User` делят один экземпляр `Address`. (Это UML-диаграмма объектов, а не классов.) Если объект `Address` должен поддерживать разделяемую ссылку во время выполнения, это – тип сущности. У экземпляра `Address` свой жизненный цикл; его нельзя удалить, когда Джон удаляет свою учетную запись (`User`), – у Джейн может оставаться ссылка на этот объект `Address`.

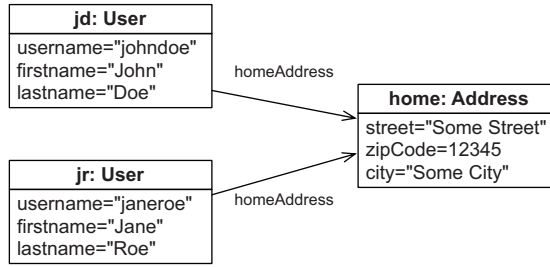


Рис. 4.1 ❖ Два экземпляра `User` ссылаются на один экземпляр `Address`

Рассмотрим теперь альтернативную модель, где каждый объект `User` ссылается на собственный экземпляр `homeAddress`, как показано на рис. 4.2. В этом случае можно сделать объект `Address` зависимым от объекта `User`: он становится типом-значением.

Когда Джон удаляет свою учетную запись (`User`), можно безопасно удалить его экземпляр `Address`. Никто больше не будет ссылаться на него.

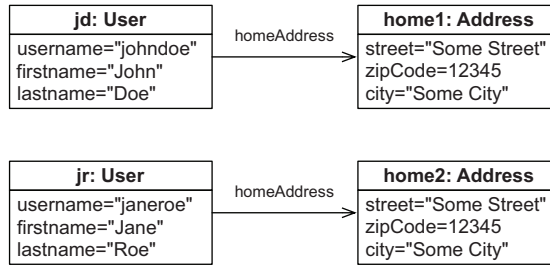


Рис. 4.2 ❖ У каждого из двух экземпляров `User` свой, зависимый экземпляр `Address`

В результате мы приходим к следующему коренному различию:

- экземпляр *типа сущности* можно извлечь, используя его хранимую идентичность – например, экземпляры `User`, `Item` или `Category`. Ссылка на экземпляр сущности (указатель в JVM) сохраняется как ссылка в базе данных (значение, ограниченное внешним ключом). У экземпляра сущности собственный жизненный цикл; он может существовать независимо от остальных сущно-

стей. Выбранные классы предметной модели отображаются как типы сущностей;

- у экземпляра *типа-значения* отсутствует хранимая идентичность; он принадлежит экземпляру сущности. Его время жизни определяется экземпляром сущности-владельца. Тип-значение не поддерживает разделения ссылок. Наиболее очевидные типы-значения – это все классы, объявленные в JDK, такие как `String`, `Integer`, а также все примитивные типы. Вы также можете отображать ваши классы модели предметной области как типы-значения – например, `Address` и `MonetaryAmount`.

Прочитав спецификацию JPA, можно обнаружить там аналогичные понятия. Только типы-значения в JPA называются *базовыми типами* (basic property types) или *встраиваемыми классами* (embeddable classes). Мы вернемся к этому в следующей главе; сейчас сфокусируемся на сущностях.

Определение классов сущностей и типов-значений в модели предметной области не происходит спонтанно, но следует определенной процедуре.

4.1.3. Разделение сущностей и типов-значений

Добавление стереотипов (механизма расширения UML) в UML-диаграмму классов иногда помогает сразу отличить сущность от типа-значения. Такой подход также заставит вас думать об этом различии для всех классов, что является первым шагом к оптимальному отображению и хорошей производительности уровня хранения. Пример показан на рис. 4.3.

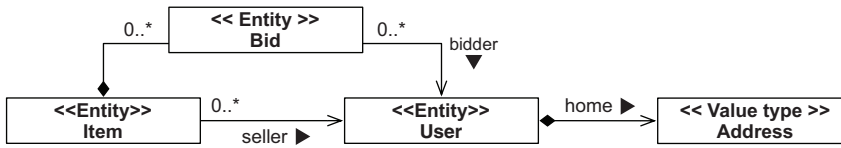


Рис. 4.3 ❖ Стереотипы сущностей и типов-значений

Очевидно, классы `User` и `Item` – сущности. У каждого своя идентичность, на их экземпляры ссылается множество других экземпляров (разделяемые ссылки), их отрезки жизни независимы.

Отметить класс `Address` как тип-значение тоже легко – один экземпляр `User` ссылается на конкретный экземпляр `Address`. Это легко понять, поскольку ассоциация представлена как композиция и экземпляр `User` целиком отвечает за жизнь экземпляра `Address`, на который он ссылается. Таким образом, больше никто не может ссылаться на данный экземпляр `Address`, и ему не нужна собственная идентичность.

Проблемы могут возникнуть с классом `Bid`. Ромбовидная ассоциация между классами `Bid` и `Item` в объектно-ориентированном моделировании называется композицией. Таким образом, объект `Item` является владельцем всех экземпляров `Bid` и хранит коллекцию ссылок. На первый взгляд это кажется разумным, ведь предложения цен за товар на аукционе бесполезны, когда товара уже нет.

Но что, если во время будущего расширения модели предметной области понадобится коллекция `User#bids`, в которой содержались бы все предложения цен, сделанные конкретным пользователем? На данном этапе ассоциация между `Bid` и `User` однонаправленная – в `Bid` есть ссылка `bidder`. Но что, если она будет двунаправленной?

В таком случае придется использовать разделяемую ссылку на экземпляры `Bid`, поэтому класс `Bid` должен быть сущностью. Хотя у него зависимый жизненный цикл, он должен обладать собственной идентичностью для поддержания (в будущем) разделяемых ссылок.

Вы часто будете сталкиваться с такими вариантами смешанного поведения; первое, что вы должны делать, – объявлять все как тип-значение и переводить класс в разряд сущностей лишь при крайней необходимости. Попробуйте упростить ассоциации – хранимые коллекции, к примеру, часто добавляют сложности, не давая никаких преимуществ. Вместо отображения коллекций `User#bids` и `Item#bids` можно было бы создать запросы для получения всех предложений цены за товар и всех предложений цены конкретного пользователя. Ассоциация на UML-диаграмме была бы направлена от класса `Bid` *в сторону* классов `Item` и `User`, и она была бы только однонаправленной. Класс `Bid` имел бы стереотип `<<Value type>>`. Мы еще вернемся к этому вопросу в главе 7.

Далее на основе диаграммы модели предметной области создаются POJO для всех сущностей и типов-значений. Необходимо обратить внимание на три вещи:

- **разделяемые ссылки** – избегайте разделяемых ссылок на экземпляры типов-значений при создании классов POJO. Например, убедитесь, что только один объект `User` может ссылаться на объект `Address`. Можно сделать класс `Address` неизменяемым, убрав общедоступный метод `setUser()`, и гарантировать целостность отношения при помощи общедоступного конструктора с аргументом `User`. Конечно, конструктор без аргументов, возможно защищенный, тоже придется объявить, чтобы, как обсуждалось в предыдущей главе, Hibernate мог создавать экземпляры;
- **зависимости жизненного цикла** – при удалении объекта `User` вместе с ним должен удаляться зависимый объект `Address`. Метаданные хранения будут включать правила каскадирования для всех подобных зависимостей, чтобы Hibernate (или база данных) могла позаботиться об удалении ненужных объектов `Address`. Соответственно, разрабатывая прикладные процедуры и пользовательский интерфейс, вы должны учитывать подобные зависимости;
- **идентичность** – в большинстве случаев классам сущностей нужно свойство-идентификатор. У типов-значений (и, конечно, классов из JDK, таких как `String` и `Integer`) поле идентификатора отсутствует, т. к. эти объекты идентифицируются через сущность-владельца.

Мы вернемся к ссылкам, ассоциациям и правилам жизненного цикла позже, когда будем обсуждать более продвинутые отображения в следующих главах. А пока нашей следующей темой будут идентичность и свойства-идентификаторы.

4.2. Отображение сущностей с идентичностью

Отображение сущностей с идентичностью требует понимания различий между равенством и идентичностью в Java. Только усвоив суть этих понятий, можно переходить к примеру с классом сущности и его отображением. После этого мы сможем углубиться в детали: выбрать первичный ключ, настроить генераторы ключей и в итоге пройтись по стратегиям генератора идентификаторов. Но прежде чем начать обсуждение таких понятий, как идентичность в *базе данных* (database identity) и как JPA управляет сущностями, разберемся с различиями между идентичностью и равенством Java-объектов.

4.2.1. Идентичность и равенство в Java

Java-разработчки понимают разницу между идентичностью и равенством Java-объектов. Понятие идентичности объектов (`==`) определяется виртуальной машиной Java. Две ссылки идентичны, если они указывают на одну область памяти.

С другой стороны, понятие равенства объектов определяется методом `equals()` класса. Равенство также часто называют *эквивалентностью*. Эквивалентность означает, что два различных (неидентичных) экземпляра имеют одно значение – одинаковое состояние. Два различных экземпляра `String` эквивалентны, если представляют одинаковую последовательность символов, даже если каждая занимает свою область памяти в виртуальной машине. (Если вы Java-гуру, то мы готовы согласиться, что `String` – это особый случай. Представьте, что мы взяли другой класс в качестве примера.)

Долговременное хранение усложняет эту картину. С точки зрения объектно-реляционной модели, хранимая сущность – это представление в памяти конкретной записи (записей) из таблицы (таблиц) базы данных. Наряду с идентичностью и равенством Java мы должны дать определение идентичности в базе данных. Теперь у вас есть три правила различения ссылок:

- объекты идентичны, если они занимают одну область памяти в JVM. Это проверяется оператором `a == b`. Это понятие известно как *объектная идентичность* (object identity);
- объекты равны, если их состояние одинаково, что определяется методом `a.equals(Object b)`. Классы, явно не переопределяющие этого метода, наследуют реализацию в `java.lang.Object`, которая сравнивает идентичность объектов с помощью `==`. Это понятие известно как *равенство объектов* (object equality);
- объекты, сохраненные в базу данных, идентичны, если находятся в одной таблице и у них одинаковый первичный ключ. На стороне Java это понятие известно как *идентичность в базе данных* (database identity).

Теперь мы должны рассмотреть, как идентичность в базе данных соотносится с объектной идентичностью и как идентичность в базе данных представляется в метаданных отображения. В качестве примера отобразим сущность предметной модели.

4.2.2. Первый класс сущности и его отображение

В предыдущей главе мы немножко слукавили – аннотации `@Entity` недостаточно для отображения хранимого класса. Вам также потребуется аннотация `@Id`, как показано в листинге 4.1.

Листинг 4.1 ❖ Отображаемый класс сущности `Item` со свойством-идентификатором

Файл: /model/src/main/java/org/jpwh/model/simple/Item.java

```
@Entity
public class Item {

    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    protected Long id;

    public Long getId() {    ← Необязательный, но полезный
        return id;
    }
}
```

Это самый простой класс сущности, отмеченный аннотацией `@Entity` как «пригодный к сохранению», с аннотацией `@Id`, отображающей свойство-идентификатор в первичный ключ в базе данных. По умолчанию класс отображается в таблицу с именем `ITEM`.

Каждый класс сущности должен иметь поле `@Id`; таким способом JPA раскрывает приложению идентичность сущностей в базе данных. Мы не показываем свойство-идентификатор на диаграммах, так как считаем, что оно есть у каждого класса сущности. В наших примерах мы всегда даем такому свойству имя `id`. Это хорошая практика; используйте одинаковое имя для свойства-идентификатора во всех классах предметной модели. Если больше ничего не указывать, это поле будет отображено в столбец первичного ключа `ID` таблицы `ITEM`.

Во время загрузки и сохранения объектов Hibernate будет обращаться к полю идентификатора напрямую, минуя методы чтения/записи. Поскольку `@Id` находится над полем класса, Hibernate по умолчанию сделает хранимым каждое поле в классе. В данном случае действует следующее правило JPA: если аннотация `@Id` расположена над полем класса, реализация JPA будет обращаться к полям класса напрямую и по умолчанию считать их все частью сохраняемого состояния. Позже в этой главе вы узнаете, как это переопределяется. По нашему опыту доступ к полям класса часто является лучшим вариантом, т. к. дает больше свободы в проектировании методов доступа.

Нужен ли (общедоступный) метод чтения для свойства-идентификатора? Вообще, приложение часто использует идентификаторы в базе данных как удобный указатель на конкретный экземпляр, даже вне уровня хранения. Например, веб-приложение обычно показывает пользователю экран результатов поиска в виде списка кратких описаний. Когда пользователь выбирает конкретный элемент, приложению может понадобиться извлечь выбранный элемент, и обычным делом

является выполнение поиска по идентификатору для этой цели. Вы наверняка уже использовали идентификаторы подобным образом даже в приложениях, использующих JDBC.

Нужен ли метод записи? Значения первичного ключа никогда не изменяются, поэтому не следует разрешать модификацию значения свойства-идентификатора. Hibernate не будет обновлять столбец первичного ключа, и вам не стоит создавать общедоступный метод записи для идентификатора сущности.

Java-тип поля идентификатора, `java.lang.Long` в предыдущем примере, зависит от типа столбца первичного ключа таблицы `ITEM` и от того, как создаются значения ключа. Это подводит нас к аннотации `@GeneratedValue` и первичным ключам в целом.

4.2.3. Выбор первичного ключа

Идентификатор сущности в базе данных отображается в первичный ключ таблицы, поэтому сначала мы должны немного узнать о первичных ключах, отложив отображения в сторону.

Потенциальный ключ (candidate key) – это столбец или множество столбцов, которые можно использовать для идентификации конкретной записи в таблице. Чтобы стать первичным ключом, потенциальный ключ должен удовлетворять следующим требованиям:

- значение любого столбца потенциального ключа не может быть `null`. Невозможно идентифицировать что-либо с помощью неизвестных данных, и в реляционной модели отсутствует значение `null`. Некоторые SQL-системы позволяют определять (составной) первичный ключ, значения столбцов которого могут быть `null`, так что будьте осторожнее;
- значение столбца (или столбцов) потенциального ключа должно быть уникальным для любой записи;
- значение столбца (или столбцов) потенциального ключа никогда не обновляется – оно неизменяемо.

Должен ли первичный ключ быть неизменяемым?

Реляционная модель утверждает, что потенциальный ключ должен быть уникальным и несократимым (никакое подмножество атрибутов ключа не может быть уникальным). Кроме того, выбор потенциального ключа в качестве первичного – это дело вкуса. Но Hibernate рассчитывает, что потенциальный ключ, выбранный в качестве первичного, неизменяем. Hibernate не позволяет обновить значение первичного ключа; попробовав обойти это требование, вы неизбежно столкнетесь с проблемами в механизмах кэширования и проверки состояния объектов в Hibernate. Если ваша схема базы данных опирается на обновляемые первичные ключи (и, возможно, использует ограничение внешнего ключа `ON UPDATE CASCADE`), вы должны поменять схему прежде, чем она станет работать с Hibernate.

Если в таблице есть лишь один идентифицирующий атрибут, он по определению является первичным ключом. Но несколько столбцов или комбинаций столбцов

также может удовлетворять этим требованиям в конкретной таблице; вы должны сделать выбор между потенциальными ключами, определив лучший первичный ключ для таблицы. Потенциальные ключи, не выбранные в качестве первичного ключа, следует объявить уникальными, если их значения действительно уникальны (но они могут быть изменяемыми).

Многие старые модели данных SQL используют естественные первичные ключи. *Естественный ключ* (natural key) – это ключ, имеющий смысл в предметной области: атрибут (или комбинация атрибутов), уникальный в силу семантики предметной области. Примерами естественных ключей могут служить номер социального страхования в США и австралийский налоговый номер. Отличить натуральный ключ довольно просто: если потенциальный ключ имеет смысл вне контекста базы данных, это натуральный ключ, независимо от того, был ли он сгенерирован автоматически. Подумайте о пользователях приложения – если они обращаются к ключевому атрибуту, когда говорят о приложении или работают с ним, это естественный ключ: «Можешь прислать мне изображение товара #123-abc?»

Опыт показывает, что рано или поздно естественные первичные ключи создают проблемы. Хороший первичный ключ должен быть уникальным, неизменяемым и никогда не иметь значения `null`. Немногие атрибуты сущностей удовлетворяют этим требованиям, а те, что удовлетворяют, не могут эффективно индексироваться базами данных SQL (хотя это деталь реализации, и она не должна быть решающим фактором за или против выбора конкретного ключа). Кроме того, нужно быть уверенным, что потенциальный ключ никогда не изменится за время существования базы данных. Изменение значения (или даже определения) первичного ключа, а также всех ссылающихся на него внешних ключей является неприятной задачей. Рассчитывайте на то, что ваша схема базы данных просуществует десятилетия, даже если само приложение – нет.

Кроме того, часто естественные ключи получаются только путем соединения нескольких столбцов в *составной* (composite) естественный ключ. Такие составные ключи, несмотря на свою полезность для некоторых артефактов схемы (таких как промежуточная таблица в отношении *многие ко многим*), потенциально могут усложнить сопровождение, выполнение произвольных запросов и эволюцию схемы. Мы продолжим обсуждение составных ключей позже, в разделе 9.2.1.

По этим причинам мы настоятельно рекомендуем использовать искусственные идентификаторы, называемые также *суррогатными ключами* (surrogate key). Суррогатные ключи не имеют смысла для предметной области – они хранят уникальные значения, сформированные базой данных или приложением. В идеале пользователи приложения не видят и не ссылаются на значения этого ключа: они являются частью внутреннего устройства системы. Добавление столбца суррогатного ключа также уместно в распространенной ситуации, когда потенциальные ключи отсутствуют. Другими словами, (почти) каждая таблица схемы должна иметь выделенный столбец суррогатного первичного ключа только для этой цели.

Существует несколько хорошо известных подходов к формированию значений суррогатного ключа. Настройка осуществляется с помощью вышеупомянутой аннотации `@GeneratedValue`.

4.2.4. Настройка генераторов ключей

Свойство-идентификатор класса сущности обязательно должно отмечаться аннотацией `@Id`. Если за ней не следует аннотация `@GeneratedValue`, реализация JPA решит, что вы берете на себя задачу формирования и присваивания значений идентификатора перед сохранением экземпляра. Мы называем это *идентификатором, назначенным приложением*. Назначение идентификаторов сущностей вручную необходимо при работе с унаследованной базой данных и/или естественными первичными ключами. Мы поговорим об этом подробнее в специальном разделе 9.2.1.

Обычно желательно, чтобы система сама формировала значение первичного ключа при сохранении экземпляра сущности, поэтому вслед за аннотацией `@Id` добавляют аннотацию `@GeneratedValue`. JPA стандартизирует несколько стратегий формирования значений, используя перечисление `javax.persistence.GenerationType`, элемент которого вы указываете в `@GeneratedValue(strategy = )`:

- `GenerationType.AUTO` – Hibernate выбирает подходящую стратегию, исходя из диалекта SQL, используемого настроенной базой данных. Это эквивалентно `@GeneratedValue()` без всяких настроек;
- `GenerationType.SEQUENCE` – Hibernate ищет (и создает, если вы используете определенные инструменты) последовательность `HIBERNATE_SEQUENCE` в базе данных. Эта последовательность будет вызываться отдельно перед каждой операцией `INSERT` для получения последовательных числовых значений;
- `GenerationType.IDENTITY` – Hibernate ищет (и создает в DDL-определении таблицы) специальный столбец первичного ключа с автоматическим приращением, который сам генерирует числовое значение во время выполнения `INSERT` в базе данных;
- `GenerationType.TABLE` – Hibernate будет использовать дополнительную таблицу в схеме базы данных, хранящую следующее числовое значение первичного ключа: по одной строке на каждый класс сущности. Соответственно, эта таблица будет читаться и обновляться перед выполнением `INSERT`. По умолчанию таблица называется `HIBERNATE_SEQUENCES` и содержит столбцы `SEQUENCE_NAME` и `SEQUENCE_NEXT_HI_VALUE`. (Внутренняя реализация использует более сложный, но эффективный алгоритм генерации идентификаторов hi/lo; подробнее об этом позже.)

Несмотря на то что `AUTO` кажется удобным вариантом, обычно требуется больший контроль, поэтому имеет смысл не полагаться на него, а настроить стратегию формирования значений первичного ключа явно. Кроме того, большинство приложений работает с последовательностями в базе данных, но вам может понадобиться поменять имя или другие настройки последовательности. Поэтому вместо выбора одной из стратегий JPA мы советуем использовать отображение идентификатора с помощью `@GeneratedValue(generator = "ID_GENERATOR")`, как показано в предыдущем примере.

Это *именованный* генератор идентификаторов; теперь можно настраивать конфигурацию `ID_GENERATOR` независимо от классов сущностей.

В JPA существуют две встроенные аннотации для настройки именованных генераторов: `@javax.persistence.SequenceGenerator` и `@javax.persistence.TableGenerator`. С их помощью можно создавать именованные генераторы с произвольными именами последовательностей и таблиц. Как обычно, JPA-аннотации можно располагать только перед классом (возможно, пустым), но не в файле *package-info.java*.

Особенность Hibernate

По этой причине, а также из-за того, что аннотации JPA не дают полного доступа ко всем особенностям Hibernate, мы предпочитаем альтернативу: оригинальную аннотацию `@org.hibernate.annotations.GenericGenerator`. Она поддерживает все стратегии генерации идентификаторов Hibernate и их настройку. В отличие от весьма ограниченных аннотаций JPA, аннотации Hibernate можно использовать в файле *package-info.java*, как правило, в том же пакете, что и классы модели предметной области. В листинге 4.2 показана рекомендованная конфигурация.

Листинг 4.2 ❖ Генератор идентификаторов Hibernate в виде метаданных уровня пакета

Файл: /model/src/main/java/org/jpwh/model/package-info.java

```
@org.hibernate.annotations.GenericGenerator(
  name = "ID_GENERATOR",
  strategy = "enhanced-sequence",     ← ❶ Стратегия применения расширенной последовательности
  parameters = {
     @org.hibernate.annotations.Parameter(
        name = "sequence_name",       ← ❷ Имя последовательности
        value = "JPWH_SEQUENCE"
     ),
     @org.hibernate.annotations.Parameter(
        name = "initial_value",       ← ❸ Начальное значение
        value = "1000"
     )
})
```

Конфигурация генератора в Hibernate имеет следующие преимущества:

- стратегия `enhanced-sequence` (❶) производит последовательные числовые значения. Если диалект SQL поддерживает последовательности, Hibernate будет использовать настоящую последовательность в базе данных. Если СУБД не поддерживает последовательности, Hibernate создаст и будет использовать дополнительную «таблицу последовательностей», имитирующую поведение последовательности. Это обеспечивает настоящую переносимость: генератор всегда будет вызываться перед выполнением SQL-инструкции `INSERT`, в отличие, например, от столбца идентификатора с автоматическим приращением, значение для которого формируется во время выполнения `INSERT` и затем возвращается приложению;
- при помощи параметра `sequence_name` (❷) можно выбрать имя последовательности. Hibernate будет использовать либо существующую последова-

тельность, либо создаст ее при автоматическом формировании SQL-схемы. Если СУБД не поддерживает последовательности, это имя станет именем специальной «таблицы последовательностей»;

- при помощи параметра `initial_value` (❸) можно указать начальное значение, что дает возможность использования тестовых данных. Например, при выполнении интеграционных тестов Hibernate будет делать любую вставку данных со значениями идентификатора больше 1000. Любые тестовые данные, которые нужно загрузить до начала теста, могут использовать числа от 1 до 999, и в тестах можно обращаться к известным значениям идентификаторов: «Загрузить товар с идентификатором 123 и выполнить тестирование с его участием». Эта настройка применяется в момент, когда Hibernate создает схему SQL и последовательность.

Одна и та же последовательность в базе данных может использоваться всеми классами предметной модели. Если указать `@GeneratedValue(generator = "ID_GENERATOR")` во всех классах сущностей, это не создаст проблем. До тех пор, пока значения первичного ключа уникальны в пределах одной таблицы, не имеет значения, следуют они друг за другом подряд или нет. Если вас беспокоит конкурентный доступ, поскольку последовательность вызывается перед каждым выполнением `INSERT`, мы обсудим другой вариант этой конфигурации генератора позже, в разделе 20.1.

Наконец, типом свойства-идентификатора класса сущности выбран `java.lang.Long`, который прекрасно отображается в числовой генератор последовательности в базе данных. Можно также использовать примитивный тип `long`. Главным отличием является возвращаемое значение метода `someItem#getId()`, еще не сохраненного в базу данных объекта: `null` или `0`. Проверка на `null`, чтобы выяснить, является объект новым или нет, возможно, будет более понятна читателю вашего кода. Не нужно использовать такие целочисленные типы, как `short` и `int`, для идентификаторов. Несмотря на то что их хватит на какое-то время (возможно, годы), по мере роста базы данных вы можете оказаться ограничены их диапазоном. Если генерировать новый идентификатор каждую миллисекунду, не делая пропусков, типа `Integer` хватит на два месяца, тогда как тип `Long` продержится 300 млн лет.

Хотя стратегия `enhanced-sequence`, показанная в листинге 4.2, является рекомендованной для большинства приложений, она не единственная, встроенная в Hibernate.

Особенность Hibernate

4.2.5. Стратегии генерации идентификаторов

Далее следует список всех стратегий генерации идентификаторов Hibernate, их параметров и наших рекомендаций по их использованию. Если сейчас вам не хочется читать весь список, выберите вариант `GenerationType.AUTO` и проверьте, что выберет Hibernate для вашего диалекта SQL по умолчанию. Вероятнее всего, это будет `sequence` или `identity` – хороший, но, возможно, не самый эффективный или

переносимый вариант. Если требуются высокая переносимость и доступность значений идентификаторов перед выполнением `INSERT`, следует использовать `enhanced-sequence`, как показано в предыдущем разделе. Это переносимая, гибкая и современная стратегия, предоставляющая также различные оптимизации для больших наборов данных.

Мы также покажем связь каждой стандартной стратегии JPA с ее эквивалентом в Hibernate. В ходе естественного развития Hibernate появились два типа отображений между стандартными и собственными стратегиями Hibernate; мы называем их в списке ниже *Старые* и *Новые* соответственно. Это отображение можно поменять с помощью настройки `hibernate.id.new_generator_mappings` в файле *persistence.xml*. Значение по умолчанию – `true`, что соответствует *Новому* отображению. Программное обеспечение стареет не как вино.

Формирование идентификаторов до или после выполнения INSERT: в чем разница?

Служба ORM старается оптимизировать SQL-инструкции `INSERT`: например, объединяя несколько запросов на уровне JDBC. То есть выполнение SQL-инструкций происходит позже в течение транзакции, а не в момент вызова `entityManager.persist(someItem)`. Этот вызов лишь добавляет операцию вставки в очередь для последующего выполнения и, если это возможно, присваивает значение идентификатора. Но если реализация не сможет сгенерировать идентификатор до выполнения инструкции `INSERT`, немедленный вызов `someItem.getId()` вернет `null`. В общем случае мы предпочитаем стратегии генерации, производящие значения идентификаторов независимо, перед выполнением `INSERT`. Общепринятым является использование разделяемой и доступной в многопоточной среде последовательности в базе данных. Столбцы с автоматическим приращением, значения столбцов по умолчанию, а также ключи, создаваемые триггерами, доступны лишь после `INSERT`.

- `native` – автоматически выбирает другую стратегию, такую как `sequence` или `identity`, в зависимости от настроенного диалекта SQL. Загляните в Javadoc (или даже в исходники) диалекта SQL, настроенного в *persistence.xml*. Аналог `GenerationType.AUTO` в JPA со старым отображением.
- `sequence` – использует последовательность с именем `HIBERNATE_SEQUENCE` в базе данных. Последовательность вызывается перед выполнением каждой инструкции `INSERT` для вставки новой записи. Вы можете поменять имя последовательности и указать дополнительные настройки DDL; см. Javadoc для класса `org.hibernate.id.SequenceGenerator`.
- `sequence-identity` – генерирует значения ключа, вызывая последовательность в базе данных во время вставки, например: `insert into ITEM(ID) values (HIBERNATE_SEQUENCE.nextval)`. Значение ключа извлекается после вставки, по аналогии со стратегией `identity`. Поддерживает те же параметры настройки, что и стратегия `sequence`; см. Javadoc для класса `org.hibernate.id.SequenceIdentityGenerator` и его родителя.

- `enhanced-sequence` – использует последовательность в базе данных, если поддерживается; в противном случае использует дополнительную таблицу с единственным столбцом и записью, имитирующую последовательность. По умолчанию таблица получает имя `HIBERNATE_SEQUENCE`. «Последовательность» всегда вызывается до выполнения `INSERT`, обеспечивая тем самым одинаковое поведение независимо от реальной поддержки последовательностей в СУБД. Поддерживает использование оптимизатора `org.hibernate.id.enhanced.Optimizer`, предотвращающего обращения к базе данных при каждой вставке: по умолчанию оптимизация отключена, и новое значение извлекается перед каждым выполнением `INSERT`. Дополнительные примеры приводятся в главе 20. Полный перечень параметров можно найти в Javadoc для класса `org.hibernate.id.enhanced.SequenceStyleGenerator`. Аналог `GenerationType.SEQUENCE` и `GenerationType.AUTO` в JPA с новым отображением – вероятно, лучший выбор среди имеющихся стратегий.
- `seqhilo` – использует оригинальную последовательность Hibernate с названием `HIBERNATE_SEQUENCE`, оптимизируя вызовы перед выполнением `INSERT` путем объединения старших (hi) и младших (lo) значений. Если полученное из последовательности значение hi равно 1, то следующие 9 вставок будут сделаны со значениями ключей 11, 12, 13, ..., 19. Затем последовательность вызывается снова для получения следующего значения hi (2 или больше), и процесс повторяется со значениями 21, 22, 23 и т. д. Максимальное значение lo (9 по умолчанию) можно настроить с помощью параметра `max_lo`. К сожалению, из-за особенностей реализации Hibernate эту стратегию *невозможно* настроить в `@GenericGenerator`. Единственная возможность ее использования – с помощью `GenerationType.SEQUENCE` в JPA и старого отображения. Настроить ее можно при помощи стандартной JPA-аннотации `@SequenceGenerator` перед (возможно, пустым) классом. Более подробную информацию о классе `org.hibernate.id.SequenceHiLoGenerator` и его родителях ищите в Javadoc. Прежде чем применять эту стратегию, подумайте о возможности применения более предпочтительной `enhanced-sequence` с оптимизатором.
- `hilo` – использует дополнительную таблицу с именем `HIBERNATE_UNIQUE_KEY` и такой же алгоритм, как в стратегии `seqhilo`. В таблице имеются один столбец и одна запись, хранящая следующее значение последовательности. По умолчанию максимальное значение lo равно 32 767, поэтому, вполне возможно, вам захочется настроить его с помощью параметра `max_lo`. Более подробную информацию о классе `org.hibernate.id.TableHiLoGenerator` ищите в Javadoc. Мы не рекомендуем эту устаревшую стратегию; используйте вместо нее `enhanced-sequence` с оптимизатором.
- `enhanced-table` – использует дополнительную таблицу `HIBERNATE_SEQUENCES`, представляющую последовательность, с одной записью по умолчанию, в которой хранится следующее значение. Это значение возвращается и обновляется, когда нужно сформировать значение идентификатора. В этой таблице можно организовать хранение нескольких записей – по одной на каждый генератор; подробности ищите в описании `org.hibernate.id.enhanced.Tabl-`

Generator в Javadoc. Аналог GenerationType.TABLE в JPA с новым отображением. Заменяет устаревший класс org.hibernate.id.MultipleHiLoPerTableGenerator, который является старым отображением для GenerationType.TABLE.

- identity – поддерживает столбцы идентификаторов (IDENTITY) и с автоматическим приращением в DB2, MySQL, MS SQL Server и Sybase. Значение идентификатора для столбца первичного ключа определяется во время вставки записи. Параметры настройки отсутствуют. К сожалению, из-за особенностей реализации Hibernate эту стратегию *невозможно* настроить в @GenericGenerator. Единственная возможность ее использования – с помощью GenerationType.IDENTITY в JPA старого или нового отображения, что делает ее стратегией по умолчанию для GenerationType.IDENTITY.
- increment – при запуске Hibernate читает максимальные (числовые) значения первичного ключа из каждой таблицы сущности и увеличивает это значение на единицу каждый раз, когда происходит вставка новой записи. Данная стратегия особенно эффективна в приложениях Hibernate, не являющихся частью кластера и имеющих эксклюзивный доступ к базе данных; не используйте эту стратегию в других ситуациях.
- select – Hibernate не будет генерировать значение ключа или добавлять столбец первичного ключа в выражение INSERT. Ожидается, что СУБД сама присвоит значение (указанное в схеме или полученное с помощью триггера) столбцу во время вставки. После вставки Hibernate извлечет столбец первичного ключа с помощью запроса SELECT. Имеет обязательный параметр key, определяющий имя свойства-идентификатора в базе данных (например, id) для выражения SELECT. Данная стратегия не очень эффективна и должна использоваться только со старыми драйверами JDBC, которые не могут возвращать сгенерированных ключей непосредственно.
- uuid32 – создает уникальный 128-битный идентификатор UUID на уровне приложения. Такая стратегия полезна, если требуется обеспечить глобальную уникальность идентификаторов в нескольких базах данных (к примеру, если каждую ночь вы объединяете данные из нескольких рабочих баз данных в архив, используя пакетную обработку). В классе сущности идентификатор UUID может быть представлен как свойство типа java.lang.String, byte[16] или java.util.UUID. Заменяет устаревшие стратегии uuid и uuid.hex. Настраивается с помощью org.hibernate.id.UUIDGenerationStrategy; подробности ищите в описании класса org.hibernate.id.UUIDGenerator в Javadoc.
- guid – использует глобально уникальный идентификатор, созданный базой данных при помощи SQL-функции, доступной в Oracle, Ingres, MS SQL Server и MySQL. Hibernate вызывает функцию базы данных перед вставкой. Значение отображается в поле идентификатора типа java.lang.String. Если потребуется более полный контроль за созданием идентификатора, настройте стратегию в аннотации @GenericGenerator, используя полное квалифицированное имя класса, реализующего интерфейс org.hibernate.id.IdentityGenerator.

Подводя итог, мы можем дать следующие рекомендации по выбору стратегии формирования идентификаторов:

- в общем случае мы предпочитаем стратегии, производящие значения идентификаторов независимо, перед выполнением `INSERT`;
- используйте `enhanced-sequence`, которая работает со встроенной последовательностью в базе данных, если та поддерживается, или с дополнительной таблицей с единственным столбцом и единственной записью, имитирующей последовательность.

Мы будем считать, что вы добавили свойство-идентификатор во все классы сущностей вашей предметной модели и что, после того как завершите создание основного отображения каждой сущности и ее свойства-идентификатора, вы продолжите отображать свойства сущностей, имеющие тип-значение. Мы обсудим отображения типов-значений в следующей главе. Читайте дальше, чтобы узнать о некоторых специальных возможностях, которые могут упростить и улучшить отображения ваших классов.

4.3. Способы отображений сущностей

Вы уже отобразили хранимый класс с помощью аннотации `@Entity`, используя значения по умолчанию для всех остальных настроек, таких как имя таблицы SQL, в которую отображается класс. В следующем разделе мы исследуем некоторые настройки, задаваемые на уровне класса, и как ими управлять:

- имена по умолчанию и стратегии именования;
- динамическое формирование SQL-запросов;
- изменяемость сущностей.

Далее обсуждаются настройки; вы можете пропустить этот раздел и вернуться позже, когда придется решать конкретную проблему.

4.3.1. Управление именами

Поговорим сначала об именовании классов сущностей и таблиц. Если хранимый класс отметить единственной аннотацией `@Entity`, по умолчанию для таблицы будет выбрано имя, совпадающее с именем класса. Обратите внимание, что имена артефактов SQL мы записываем `ПРОПИСНЫМИ` буквами, чтобы их было легче различать, – SQL на самом деле не чувствителен к регистру. Таким образом, Java-класс `Item` отображается в таблицу `ITEM`. Вы можете переопределить имя таблицы, используя JPA-аннотацию `@Table`, как показано далее.

Листинг 4.3 ❖ Аннотация `@Table` переопределяет имя таблицы

Файл: /model/src/main/java/org/jpwh/model/simple/User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {
    // ...
}
```

По умолчанию сущность `User` была бы отображена в таблицу `USER`, однако во многих СУБД SQL это имя зарезервировано. Поскольку нельзя создать таблицу с таким именем, можно отобразить класс в таблицу `USERS`. Аннотация `@javax.persistence.Table` имеет также параметры `catalog` и `schema`, если конфигурация вашей базы данных требует их наличия в качестве префиксов имени.

Заключение в кавычки позволяет при необходимости использовать зарезервированные имена SQL и даже использовать имена, чувствительные к регистру.

Заключение идентификаторов SQL в кавычки

Время от времени, особенно в старых базах данных, можно встретить имена со странными символами или пробелами, или возникает потребность учитывать регистр символов. Или, как в предыдущем примере, автоматическое отображение класса или поля может приводить к использованию зарезервированного имени таблицы или столбца.

Hibernate 5 узнает зарезервированные слова вашей СУБД по настроенному диалекту базы данных и может автоматически ставить кавычки вокруг таких строк во время формирования SQL, активировать автоматическое заключение в кавычки с помощью параметра `hibernate.auto_quote_keyword=true` в конфигурации единицы хранения. Если вы используете старую версию Hibernate или обнаружили, что информации о диалекте недостаточно, при возникновении конфликтов с ключевыми словами вам придется вручную добавить кавычки вокруг имен в ваших отображениях.

Если заключить имя таблицы или столбца в отображении в обратные кавычки, Hibernate всегда будет заключать этот идентификатор в кавычки во время формирования SQL. Этот прием продолжает действовать в последних версиях Hibernate, но спецификация JPA 2.0 стандартизовала эту функциональность как *отделяемый идентификатор* (delimited identifier) с двойными кавычками.

Вот вариант предыдущего примера с обратными кавычками, доступный только в Hibernate:

```
@Table(name = "`USER`")
```

Для совместимости с JPA необходимо также экранировать кавычки внутри строки:

```
@Table(name = "\"USER\"")
```

Любой из этих вариантов будет работать с Hibernate. Фреймворк знает, какие кавычки применять для вашего диалекта, и будет генерировать соответствующий SQL: `[USER]` для MS SQL Server, `'USER'` для MySQL, `"USER"` для H2 и т. д.

Если потребуется заключить в кавычки *каждый* SQL-идентификатор – создайте файл *orm.xml* и добавьте элемент `<delimited-identifiers/>` в раздел `<persistence-unit-defaults>`, как показано в листинге 3.8. В этом случае Hibernate будет повсеместно использовать идентификаторы, заключенные в кавычки.

Обязательно подумайте о возможности переименования таблиц и столбцов, имена которых совпадают с зарезервированными словами. Трудно писать про-

извольные SQL-запросы в консоли SQL, когда приходится вручную расставлять и экранировать кавычки.

Далее вы увидите, как Hibernate может помочь в случае, когда имеются строгие соглашения об именовании таблиц и столбцов.

Особенность Hibernate

Реализация соглашений об именовании

Hibernate предоставляет возможность автоматического соблюдения стандартов именования. Предположим, что все таблицы приложения CaveatEmptor должны следовать шаблону `CE_<table name>`. Для этого можно вручную добавить аннотацию `@Table` перед каждым классом сущности. Такой подход требует времени, и при его использовании легко что-нибудь пропустить. Вместо этого можно реализовать Hibernate-интерфейс `PhysicalNamingStrategy` или переопределить существующую реализацию, как показано в листинге 4.4.

Листинг 4.4 ❖ Реализация `PhysicalNamingStrategy`, переопределяющая соглашения об именовании по умолчанию

Файл: /shared/src/main/java/org/jpwh/shared/CENamingStrategy.java

```
public class CENamingStrategy extends
    org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl {

    @Override
    public Identifier toPhysicalTableName(Identifier name,
                                          JdbcEnvironment context) {
        return new Identifier("CE_" + name.getText(), name.isQuoted());
    }

}
```

Переопределенный метод `toPhysicalTableName()` добавляет приставку `CE_` к каждому сгенерированному имени таблицы в вашей схеме. Загляните в Javadoc с описанием интерфейса `PhysicalNamingStrategy` – он определяет также методы для произвольного именования столбцов, последовательностей и прочих артефактов.

После реализации стратегии именования ее следует активировать в *persistence.xml*:

```
<persistence-unit>name="CaveatEmptorPU">
    ...
    <properties>
        <property name="hibernate.physical_naming_strategy"
                  value="org.jpwh.shared.CENamingStrategy"/>
    </properties>
</persistence-unit>
```

Второй способ формирования нестандартных имен заключается в использовании интерфейса `ImplicitNamingStrategy`. В отличие от физической стратегии име-

нования PhysicalNamingStrategy, которая включается в работу на самом нижнем уровне, когда формируются окончательные имена артефактов, неявная стратегия именования ImplicitNamingStrategy вызывается раньше. Если отображается класс сущности, не имеющий аннотации @Table с явным именем таблицы, выбор имени производится неявной стратегией. На результат оказывают влияние такие факторы, как имя сущности и имя класса. Hibernate включает несколько готовых стратегий для формирования устаревших или совместимых с JPA имен. По умолчанию используется стратегия ImplicitNamingStrategyJpaCompliantImpl.

Давайте взглянем на другую схожую проблему – именование сущностей для запросов.

Именование сущностей для запросов

По умолчанию имена всех сущностей автоматически импортируются в пространство имен движка запросов. Иными словами, в запросах JPA можно использовать короткие имена классов без префикса с именем пакета, что очень удобно:

```
List result = em.createQuery("select i from Item i")
                .getResultList();
```

Однако этот прием работает, только когда в единице хранения имеется единственный класс Item. Если добавить еще один класс Item в другой пакет, один из них придется переименовать, чтобы продолжать использовать короткие имена в JPA-запросах:

```
package my.other.model;

@javax.persistence.Entity(name = "AuctionItem")
public class Item {
    // ...
}
```

Короткая форма запроса для класса Item из пакета my.other.model теперь будет выглядеть так: select i from AuctionItem i. Таким способом мы разрешили конфликт имен с классом Item из другого пакета. Конечно, всегда можно использовать полное квалифицированное имя с префиксом из имени пакета.

Мы завершаем наш обзор вариантов именования в Hibernate. Далее мы обсудим, как Hibernate формирует код SQL, содержащий эти имена.

Особенность Hibernate

4.3.2. Динамическое формирование SQL

По умолчанию Hibernate формирует SQL-выражения для каждого хранимого класса в момент создания единицы хранения на запуске. Эти выражения представляют простые операции (CRUD) создания (create), чтения (read), изменения (update), удаления (delete): для чтения одной записи, удаления записи и т. д. Выгоднее постоянно хранить эти запросы в памяти, а не генерировать их каждый раз,

когда во время работы приложения потребуется выполнить тот или иной запрос. Кроме того, кэширование подготовленных операторов на уровне JDBC тем более эффективно, чем меньше имеется операторов.

Как может Hibernate создавать выражения `UPDATE` при запуске? В конце концов, столбцы для изменения в этот момент неизвестны. Ответ прост: сгенерированное SQL-выражение обновляет все столбцы, и если значение какого-либо столбца не изменилось, выражение присваивает ему старое значение.

В некоторых случаях, например при работе со старыми таблицами с сотнями столбцов, когда SQL-выражения могут быть большими даже для простейших операций (например, когда требуется обновить всего один столбец), вы должны отключить формирование SQL при запуске и переключиться на динамическое формирование SQL-выражений во время работы приложения. Чрезмерно большое количество сущностей также может повлиять на время запуска, потому что сначала Hibernate должен сгенерировать все SQL-выражения для операций CRUD. Потребление памяти кэшем запросов также будет велико, если туда поместить десятки выражений для тысяч сущностей. Это может стать проблемой в виртуальном окружении с ограничениями памяти или на устройствах малой мощности.

Чтобы деактивировать создание SQL-выражений `INSERT` и `UPDATE` при запуске, потребуются оригинальные аннотации Hibernate:

```
@Entity
@org.hibernate.annotations.DynamicInsert
@org.hibernate.annotations.DynamicUpdate
public class Item {
    // ...
}
```

Активируя динамическое создание инструкций вставки и изменения, вы сообщаете Hibernate, что строки SQL должны формироваться по требованию, а не заранее. В этом случае инструкция `UPDATE` будет содержать только столбцы с новыми значениями, а `INSERT` – только столбцы, которые не могут принимать значение `null`.

Мы еще вернемся к теме формирования и модификации SQL в главе 17. Иногда вы можете вообще избежать формирования выражений `UPDATE`, если ваши сущности неизменяемы.

Особенность Hibernate

4.3.3. Неизменяемые сущности

Экземпляры конкретного класса могут быть неизменяемыми. Например, в приложении CaveatEmptor предложение цены за товар (`Bid`) не изменяется. Поэтому Hibernate никогда не потребуется выполнять операцию `UPDATE` над таблицей `BID`. Hibernate также может выполнить несколько оптимизаций – например, не проверять состояния объекта, когда отображается неизменяемый класс, как показано

в следующем примере. Здесь класс `Bid` – неизменяемый, и его экземпляры никогда не обновляются:

```
@Entity
@org.hibernate.annotations.Immutable
public class Bid {
    // ...
}
```

POJO считается неизменяемым, если не имеет общедоступных методов записи ни для каких свойств – все значения устанавливаются в конструкторе. Hibernate должен обращаться к полям класса напрямую во время загрузки и сохранения экземпляров. Как уже обсуждалось выше в этой главе: если поле класса отмечено аннотацией `@Id`, Hibernate будет обращаться к полям класса напрямую, а вы можете проектировать свои методы доступа, как считаете нужным. Но помните, что не все фреймворки могут работать с POJO в отсутствие методов записи; JSF, например, не обращается напрямую к полям класса для формирования состояния объекта.

Когда нет возможности создать представление в схеме базы данных, неизменяемый класс сущности можно отобразить в SQL-запрос `SELECT`.

Особенность Hibernate

4.3.4. Отображение сущности в подзапрос

Бывает, что администратор базы данных не позволяет менять схему базы данных; невозможно добавить даже одно представление. Давайте представим, что вы хотите создать представление, показывающее идентификатор аукционного товара (`Item`) и количество всех предложений цены за него.

Используя аннотации Hibernate, можно создать представление уровня приложения – класс сущности с доступом только на чтение, отображаемый в SQL-запрос `SELECT`:

Файл: /model/src/main/java/org/jpwh/model/advanced/ItemBidSummary.java

```
@Entity
@org.hibernate.annotations.Immutable
@org.hibernate.annotations.Subselect(
    value = "select i.ID as ITEMID, i.ITEM_NAME as NAME, " +
            "count(b.ID) as NUMBEROFBIDS " +
            "from ITEM i left outer join BID b on i.ID = b.ITEM_ID " +
            "group by i.ID, i.ITEM_NAME"
)
@org.hibernate.annotations.Synchronize({"Item", "Bid"})
public class ItemBidSummary {

    @Id
    protected Long itemId;
```

TODO: имена таблиц чувствительны к регистру (баг Hibernate HHH-8430)

```
    protected String name;

    protected long numberOfBids;

    public ItemBidSummary() {
    }

        // Методы чтения...

        // ...
}
```

При загрузке экземпляра `ItemBidSummary` Hibernate выполнит SQL-запрос `SELECT` как подзапрос:

Файл: /examples/src/test/java/org/jpwh/test/advanced/MappedSubselect.java

```
ItemBidSummary itemBidSummary = em.find(ItemBidSummary.class, ITEM_ID);
// select * from (
//      select i.ID as ITEMID, i.ITEM_NAME as NAME, ...
// ) where ITEMID = ?
```

Все имена таблиц, упомянутые в запросе `SELECT`, должны быть перечислены в аннотации `@org.hibernate.annotations.Synchronize`. (На момент написания книги в Hibernate имелась ошибка с номером HHH-8430[1], которая делала имена синхронизируемых таблиц чувствительными к регистру.) Тогда Hibernate будет знать, что перед выполнением запроса к `ItemBidSummary` необходимо синхронизировать изменения в `Item` и `Bid` с базой данных.

Файл: /examples/src/test/java/org/jpwh/test/advanced/MappedSubselect.java

```
Item item = em.find(Item.class, ITEM_ID);
item.setName("New name");

// ItemBidSummary itemBidSummary = em.find(ItemBidSummary.class, ITEM_ID);

Query query = em.createQuery(
    "select ibs from ItemBidSummary ibs where ibs.itemId = :id"
);
ItemBidSummary itemBidSummary =
    (ItemBidSummary)query.setParameter("id", ITEM_ID).getSingleResult();
```

При загрузке по идентификатору синхронизация не происходит

Автоматическая синхронизация происходит перед выполнением запроса, только когда задействованы синхронизируемые таблицы

Обратите внимание, что Hibernate не выполняет автоматической синхронизации перед операцией `find()` – только перед выполнением запроса (`Query`), если это необходимо. Hibernate определит, что модифицированный объект `Item` повлияет на результат запроса, потому что таблица `ITEM` синхронизирована с `ItemBidSummary`. Соответственно, необходимо выполнить синхронизацию изменений и инструкцию `UPDATE`, чтобы предотвратить возвращение запросом устаревших данных.

[1] См. https://hibernate.atlassian.net/browse/HHH-8430.

4.4. Резюме

- Сущности – это укрупненные классы системы. Их экземпляры обладают независимым жизненным циклом и собственной идентичностью, и на них может ссылаться множество других объектов.
- Типы-значения, с другой стороны, зависят от конкретного класса сущности. Экземпляр типа-значения связан с экземпляром сущности-владельца, и только один объект может на него ссылаться – у него отсутствует собственная идентичность.
- Мы рассмотрели понятия идентичности и равенства в Java, идентичности в базе данных, а также узнали, что такое хороший первичный ключ. Вы узнали, какие генераторы значений первичного ключа поддерживаются фреймворком Hibernate «из коробки» и как использовать и расширить систему идентификации.
- Мы обсудили некоторые полезные способы отображений классов, такие как стратегии именования и динамическое формирование SQL.

Глава 5

Отображение типов-значений

В этой главе:
- отображение полей основных типов;
- отображение встраиваемых компонентов;
- управление отображением между типами Java и SQL.

Посвятив практически всю предыдущую главу сущностям и соответствующим настройкам отображений классов и идентификаторов, сейчас мы сосредоточимся на типах-значениях разных видов. Мы поделим типы-значения на две категории: основные классы типов-значений из JDK, такие как `String`, `Date`, примитивные типы и их обертки; и типы-значения, определяемые разработчиком, такие как `Address` и `MonetaryAmount` в приложении CaveatEmptor.

В этой главе мы сначала отобразим хранимые поля с типами из JDK и познакомимся с основными аннотациями. Вы узнаете, как управлять различными аспектами свойств: переопределять значения по умолчанию, настраивать доступ и работать со сгенерированными значениями. Также вы увидите применение SQL для производных свойств и преобразованных значений столбцов. Знакомство с основными типами мы завершим обзором свойств для представления времени и отображением перечислений. Затем мы обсудим пользовательские классы типов-значений и отобразим их как встраиваемые компоненты. Вы узнаете, как классы соотносятся со схемой базы данных, и сделаете ваши классы встраиваемыми, позволяя в то же время переопределять встроенные атрибуты. Знакомство со встраиваемыми компонентами мы завершим отображением вложенных компонентов. Наконец, мы обсудим более низкоуровневую настройку загрузки и сохранения значений полей с применением гибких конвертеров JPA – стандартного механизма расширения реализаций JPA.

Главные нововведения в JPA 2

- Возможность переключения между прямым доступом к полям и через методы чтения/записи свойств при помощи аннотации `@Access`.

- Создание нескольких вложенных уровней встраиваемых компонентов и возможность применения аннотации `@AttributeOverride` ко вложенным встроенным полям с применением точечной нотации.
- Добавлена поддержка конвертеров для атрибутов основных типов, позволяющая управлять загрузкой и сохранением значений, а также их преобразованием.

5.1. Отображение полей основных типов

При отображении хранимого класса, будь то сущность или встраиваемый тип (подробнее об этом далее в разделе 5.2), каждое его свойство по умолчанию считается хранимым. К свойствам хранимого класса по умолчанию применяются следующие правила JPA:

- если свойство имеет простой тип, или это обертка простого типа, или это тип `String`, `BigInteger`, `BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `byte[]`, `Byte[]`, `char[]` или `Character[]`, оно автоматически становится хранимым. Hibernate загружает и сохраняет значение свойства в столбце соответствующего типа SQL и с именем свойства;
- иначе, если класс отмечен аннотацией `@Embeddable` или само свойство отображается как `@Embedded`, Hibernate отобразит свойство как встроенный компонент класса-владельца. Мы обсудим встраивание компонентов далее в этой главе на примере встраиваемых классов `Address` и `MonetaryAmount` в приложении CaveatEmptor;
- иначе, если свойство имеет тип `java.io.Serializable`, значение сохраняется в сериализованной форме. Часто это не то, что нужно, и лучше всегда отображать Java-классы вместо хранения кучи байтов в базе данных. Представьте, каково будет сопровождать базу данных с двоичной информацией по прошествии нескольких лет;
- иначе в момент запуска Hibernate возбудит исключение, сообщая о неизвестном типе свойства.

Согласно такому подходу, известному как *конфигурация исключением* (configuration by exception), не требуется отмечать свойство аннотацией, чтобы сделать его хранимым; настраивать отображение придется лишь в крайнем случае. В JPA имеется несколько аннотаций для управления и настройки отображений свойств основных типов.

5.1.1. Переопределение настроек по умолчанию для свойств основных типов

Иногда не требуется делать хранимыми все свойства класса сущности. Например, определенно имеет смысл сделать хранимым свойство `Item#initialPrice`, но свойство `Item#totalPriceIncludingTax` не должно быть хранимым, потому что его значение вычисляется и используется только во время выполнения и, следовательно, не должно сохраняться в базе данных.

Чтобы исключить свойство, отметьте поле или его метод чтения аннотацией `@java.persistence.Transient` или используйте ключевое слово `transient`. Обычно ключевое слово `transient` исключает поля только при сериализации в Java, но оно также распознается реализациями JPA.

Мы вернемся к аннотированию полей классов или методов чтения через минуту. А пока давайте, как и ранее, предположим, что Hibernate будет обращаться к полям класса напрямую из-за наличия аннотации `@Id` над некоторым полем класса. То есть все остальные аннотации отображения JPA и Hibernate также располагаются над полями класса.

Если вы не хотите полагаться на отображение свойств по умолчанию, примените к конкретному свойству аннотацию `@Basic` – например, к свойству `InitialPrice` класса `Item`:

```
@Basic(optional = false)
BigDecimal initialPrice;
```

Следует признать, что от этой аннотации мало пользы. У нее только два параметра: тот, что показан здесь, `optional`, отмечает свойство как необязательное на уровне Java-объекта. По умолчанию все хранимые свойства необязательны и могут иметь значение null: у товара (`Item`) может быть неизвестна начальная цена (`initialPrice`). Отображение свойства `initialPrice` как обязательного имеет смысл, если столбец `INITIALPRICE` имеет ограничение `NOT NULL` в схеме SQL. При создании SQL-схемы Hibernate автоматически добавит ограничение `NOT NULL` для каждого обязательного поля.

Если теперь попробовать сохранить экземпляр `Item`, забыв присвоить значение свойству `initialPrice`, Hibernate возбудит исключение еще до того, как попытается выполнить SQL-запрос в базе данных. Hibernate знает, что для выполнения операций `INSERT` или `UPDATE` требуется значение. Если вы не отметили поле как обязательное и попробуете сохранить `NULL`, база данных отклонит SQL-инструкцию и Hibernate возбудит исключение нарушения ограничений. Разница в конечном итоге несущественная, но лучше избегать обращения к базе данных с инструкцией, которая заведомо не сработает. Другой параметр аннотации `@Basic` – `fetch` – мы обсудим позже, когда будем разбирать стратегии оптимизации в разделе 12.1.

Чтобы объявить о возможности присвоить значение null, большинство инженеров вместо аннотации `@Basic` использует более универсальную аннотацию `@Column`:

```
@Column(nullable = false)
BigDecimal initialPrice;
```

Мы показали вам три способа объявления обязательного наличия значения в свойстве: с использованием аннотации `@Basic`, аннотации `@Column` и при помощи аннотации `@NotNull` из спецификации Bean Validation, представленной в разделе 3.3.2. Все они влияют на реализацию JPA одинаково: Hibernate делает проверку на `null` при сохранении и генерирует ограничение `NOT NULL` в схеме базы данных. Мы рекомендуем использовать аннотацию `@NotNull` из спецификации Bean Valida-

tion, чтобы можно было вручную проверять экземпляр класса `Item` и/или чтобы код интерфейса на уровне представления мог автоматически осуществлять валидацию.

С помощью аннотации `@Column` также можно переопределить имя столбца в базе данных:

```
@Column(name = "START_PRICE", nullable = false)
BigDecimal initialPrice;
```

Аннотация `@Column` имеет еще несколько параметров, большинство из которых управляет такими аспектами уровня SQL, как имя `каталога` и `схемы`. Они редко используются, и мы будем демонстрировать их только при необходимости.

Аннотации свойств не всегда располагаются над полями, и вам, возможно, не хочется, чтобы Hibernate обращался к полям напрямую.

5.1.2. Настройка доступа к свойствам

Механизм хранения обращается к свойствам класса либо напрямую, через поля, либо косвенно, через методы доступа. Для аннотированной сущности используется доступ по умолчанию, в зависимости от местоположения обязательной аннотации `@Id`. Например, если поместить `@Id` над полем, а не над методом чтения, все остальные аннотации отображения в этой сущности должны применяться к полям. Аннотации никогда не размещаются над методами записи.

Стратегия доступа по умолчанию может применяться не только к единственному классу сущности. Любой встроенный класс, отмеченный аннотацией `@Embedded`, наследует либо стратегию доступа по умолчанию, либо определенную явно стратегию корневого класса-владельца. Мы рассмотрим встроенные компоненты далее в этой главе. Кроме того, Hibernate обращается к любым свойствам с аннотацией `@MappedSuperclass`, используя либо стратегию доступа по умолчанию, либо определенную явно стратегию отображаемого класса сущности. Наследование будет темой главы 6.

Спецификация JPA определяет аннотацию `@Access` для переопределения поведения по умолчанию при помощи параметров `AccessType.FIELD` и `AccessType.PROPERTY`. Если поместить аннотацию `@Access` на уровне класса/сущности, Hibernate будет обращаться ко всем свойствам класса в соответствии с выбранной стратегией. После этого все аннотации отображения, включая `@Id`, должны помещаться либо надо полями, либо над методами чтения соответственно.

Для переопределения стратегии доступа к отдельным свойствам можно использовать аннотацию `@Access`. Давайте рассмотрим это на примере.

Листинг 5.1 ❖ Переопределение стратегии доступа для свойства `name`

Файл: /model/src/main/java/org/jpwh/model/advanced/Item.java

```
@Entity
public class Item {

    @Id                                    ← ❶ Аннотация @Id находится над полем
```

```
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @Access(AccessType.PROPERTY)   ← ❷ Переключает способ доступа к свойству во время выполнения
    @Column(name = "ITEM_NAME")    ← Аннотации должны находиться здесь!
    protected String name;

    public String getName() {      ← ❸ Вызывается при загрузке/сохранении
        return name;
    }

    public void setName(String name) {
        this.name =
            !name.startsWith("AUCTION: ") ? "AUCTION: " + name : name;
    }
}
```

❶ Для сущности `Item` по умолчанию используется прямой доступ к полям. Аннотация `@Id` расположена над полем. (Чреватая опечатками строка `ID_GENERATOR` стала константой.)
❷ Параметр `@Access(AccessType.PROPERTY)` над полем `name` включает режим доступа к этому конкретному свойству через методы чтения/записи.
❸ Во время загрузки и сохранения объектов Hibernate будет вызывать методы `getName()` и `setName()`.

Обратите внимание, что расположение других аннотаций отображения, таких как `@Column`, не изменилось – изменился лишь способ доступа к экземплярам во время выполнения.

Можно посмотреть на это с другой стороны: если для сущности по умолчанию (или явно) задан доступ к свойствам через методы чтения/записи, аннотация `@Access(AccessType.FIELD)` над методом чтения заставит Hibernate обращаться к полю напрямую. Вся остальная информация об отображении по-прежнему должна располагаться над методом чтения, а не над полем.

Особенности Hibernate

Hibernate включает редко используемое расширение: тип доступа к свойству под названием `noop` («no operations»). Звучит странно, но это позволяет ссылаться на виртуальные свойства в запросах. Это может пригодиться, когда в базе данных есть столбец, который хотелось бы использовать только в запросах JPA. Например, предположим, что в таблице `ITEM` есть столбец `VALIDATED` и ваше Hibernate-приложение не должно обращаться к этому столбцу посредством предметной модели. Это может быть столбец из унаследованной схемы или управляемый другим приложением или триггером в базе данных. Вам требуется, чтобы обращение к этому столбцу было возможно только в JPA-запросах, таких как `select i from Item i where i.validated = true` или `select i.id, i.validated from Item i`. В Java-классе `Item` подобное поле отсутствует, как следствие для аннотации нет подходящего места. Единственная возможность отображения такого виртуального поля – с помощью оригинального файла метаданных *hbm.xml*:

```
<hibernate-mapping>
    <class name="Item">
        <id name="id">
            ...
        </id>
        <property name="validated"
                  column="VALIDATED"
                  access="noop"/>
    </class>
</hibernate-mapping>
```

Это отображение сообщает Hibernate, что в запросах вы хотели бы получать доступ к виртуальному полю `Item#validated`, отображаемому в столбец `VALIDATED`, но чтобы во время выполнения операций с экземпляром `Item` не выполнялось никаких операций чтений/записи этого значения. В классе этот атрибут отсутствует. Помните, что такой файл должен целиком и полностью определять все параметры отображения – любые аннотации в классе `Item` будут игнорироваться!

Если ни одна из встроенных стратегий доступа не подходит, можно определить собственную, модифицированную стратегию доступа к свойствам, реализуя интерфейс `org.hibernate.property.PropertyAccessor`. Включите настраиваемый доступ с помощью квалифицированного имени в аннотации Hibernate: `@org.hibernate.annotations.AttributeAccessor("my.custom.Accessor")`. Обратите внимание, что аннотация `AttributeAccessor` появилась в Hibernate 4.3 как замена устаревшей `org.hibernate.annotations.AccessType`, которую было легко перепутать с JPA-перечислением `javax.persistence.AccessType`.

Некоторые свойства не отображаются в столбцы. В частности, вычисляемым свойствам присваиваются значения, получаемые из SQL-выражений.

Особенности Hibernate

5.1.3. Работа с вычисляемыми полями

Значение вычисляемого свойства – это результат SQL-выражения, объявленного при помощи аннотации `@org.hibernate.annotations.Formula`; см. листинг 5.2.

Листинг 5.2 ❖ Два вычисляемых поля, доступных только для чтения

```
@org.hibernate.annotations.Formula(
    "substr(DESCRIPTION, 1, 12) || '...'"
)
protected String shortDescription;

@org.hibernate.annotations.Formula(
    "(select avg(b.AMOUNT) from BID b where b.ITEM_ID = ID)"
)
protected BigDecimal averageBidAmount;
```

Приведенные SQL-формулы вычисляются каждый раз при загрузке сущности `Item` из базы данных и ни в какое другое время, поэтому результат может оказаться устаревшим, если другие поля изменятся. Эти поля никогда не участвуют в SQL-операциях `INSERT` или `UPDATE`, только в `SELECT`. Вычисление происходит в базе данных; Hibernate добавляет SQL-формулу в предложение `SELECT` при загрузке экземпляра.

Формулы могут ссылаться на столбцы таблицы базы данных, вызывать функции SQL и даже содержать подзапросы SQL. В предыдущем примере вызывается функция `SUBSTR()`, а также используется оператор конкатенации `||`. Выражение SQL передается в базу данных как есть; не проявив должной осторожности, положившись на операторы или ключевые слова конкретной реализации, вы свяжете отображение метаданных с конкретной базой данных. Обратите внимание, что неквалифицированные имена ссылаются на таблицу класса, которому принадлежит вычисляемое поле.

База данных вычисляет SQL-выражения в формулах, только когда Hibernate извлекает экземпляр сущности из базы данных. Hibernate также поддерживает разновидность формул, называемую *преобразователями столбцов* (column transformers), которые позволяют использовать произвольное SQL-выражение для чтения *и* записи значения свойства.

Особенности Hibernate

5.1.4. Преобразование значений столбцов

Предположим, что в базе данных есть столбец с именем `IMPERIALWEIGHT`, в котором хранится вес товара (`Item`) в фунтах. В приложении, однако, имеется свойство `Item#metricWeight`, отражающее вес в килограммах, поэтому приходится преобразовывать значение столбца базы данных при чтении *и* при записи строки в таблицу `ITEM`. Эту операцию можно реализовать с помощью расширения Hibernate – аннотации `@org.hibernate.annotations.ColumnTransformer`.

Листинг 5.3 ❖ Преобразование значений столбца с помощью SQL-выражений

```
@Column(name = "IMPERIALWEIGHT")
@org.hibernate.annotations.ColumnTransformer(
    read = "IMPERIALWEIGHT / 2.20462",
    write = "? * 2.20462"
)
protected double metricWeight;
```

При чтении записи из таблицы `ITEM` Hibernate использует выражение `IMPERIALWEIGHT / 2.20462`, следовательно, вычисление будет выполнено в базе данных, и в итоге Hibernate вернет уровню приложения значение в метрической системе. Перед записью в столбец Hibernate подставит метрическое значение на место единственного обязательного параметра (знака вопроса), и выражение SQL вычислит действительное значение для вставки или изменения.

Hibernate также применяет преобразователи столбцов в ограничениях запросов. К примеру, следующий запрос извлекает все объекты с весом в два килограмма:

```
List<Item> result =
    em.createQuery("select i from Item i where i.metricWeight = :w")
        .setParameter("w", 2.0)
        .getResultList();
```

В действительности SQL, который Hibernate применит для этого запроса, будет содержать следующее ограничение в предложении `WHERE`:

```
// ...
where
    i.IMPERIALWEIGHT / 2.20462=?
```

Обратите внимание, что для этого ограничения база данных, скорее всего, не сможет использовать индекс, в результате произойдет полное сканирование таблицы, потому что для проверки ограничения вес должен быть рассчитан в *каждой* записи таблицы `ITEM`.

Есть другой, особый тип поля, который полагается на значения, сформированные базой данных.

Особенности Hibernate

5.1.5. Значения свойств, генерируемые по умолчанию

Иногда значения свойств генерирует база данных – как правило, при вставке строки. Примерами таких значений являются отметки времени создания, цена товара по умолчанию, а также триггер, срабатывающий при каждом изменении.

Обычно приложения Hiberante должны обновлять экземпляры, содержащие любые свойства, которым база данных присваивает значения после сохранения. То есть чтобы прочитать значение после вставки или изменения записи, необходимо повторно обратиться к базе данных. Но если отметить свойства как генерируемые базой данных, приложение переложит эту работу на Hibernate. По сути, каждый раз, когда Hibernate выполняет SQL-выражение `INSERT` или `UPDATE` для сущности с генерируемыми свойствами, он тотчас же после этого выполняет `SELECT` для получения их значений.

Генерируемые свойства отмечаются аннотацией `@org.hibernate.annotations.Generated`.

Листинг 5.4 ❖ Генерирование значений свойств на уровне базы данных

```
@Temporal(TemporalType.TIMESTAMP)
@Column(insertable = false, updatable = false)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
protected Date lastModified;
```

```
@Column(insertable = false)
@org.hibernate.annotations.ColumnDefault("1.00")
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
protected BigDecimal initialPrice;
```

Аннотация `GenerationTime` имеет всего два параметра: `ALWAYS` и `INSERT`.

При выборе параметра `ALWAYS` Hibernate будет обновлять экземпляр сущности после каждой SQL-операции `UPDATE` или `INSERT`. В примере предполагается, что триггер базы данных поддерживает свойство `lastModified` в актуальном состоянии. Свойство также должно быть отмечено как доступное только для чтения при помощи параметров `updatable` и `insertable` аннотации `@Column`. Если обоим присвоено значение `false`, столбец, соответствующий этому свойству, никогда не появится в выражениях `INSERT` и `UPDATE`: генерировать значение будет база данных.

При выборе `GenerationTime.INSERT` изменение производится только после SQL-операции `INSERT`, чтобы извлечь из базы данных значение по умолчанию. Hibernate также отобразит это поле как недоступное для операции вставки. Аннотация Hibernate `@ColumnDefault` устанавливает значение столбца по умолчанию, когда Hibernate экспортирует или формирует DDL схемы SQL.

Отметки времени, как правило, генерируются автоматически базой данных или, как в предыдущем примере, приложением. Давайте поближе познакомимся с аннотацией `@Temporal`, представленной в листинге 5.4.

5.1.6. Свойства для представления времени

Свойство `lastModified` в последнем примере имеет тип `java.util.Date`, и его значение генерируется триггером в базе данных при выполнении SQL-операции `INSERT`. Спецификация JPA требует, чтобы свойства, представляющие время, отмечались аннотацией `@Temporal` для более точного определения SQL-типов данных столбцов, в которые происходит отображение. В Java для представления времени используются типы `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time` и `java.sql.Timestamp`. Hibernate также поддерживает классы из пакета `java.time`, доступного в JDK 8. (На самом деле аннотации не нужны, если к свойству применяется или может применяться *конвертер*. Вы встретитесь с конвертерами позже в этой главе.)

В листинге 5.5 показан совместимый с JPA пример – типичное свойство с отметкой времени – «этот объект был создан тогда-то», – которое сохраняется единожды и больше не обновляется никогда.

Листинг 5.5 ❖ Свойство для представления времени, которое нужно отметить аннотацией `@Temporal`

```
@Temporal(TemporalType.TIMESTAMP)
@Column(updatable = false)
@org.hibernate.annotations.CreationTimestamp
protected Date createdOn;
```

Спецификация JPA требует использовать аннотацию @Temporal. Без нее Hibernate будет использовать тип по умолчанию TIMESTAMP

```
// Java 8 API
// protected Instant reviewedOn;
```

Доступными вариантами `TemporalType` являются `DATE`, `TIME` и `TIMESTAMP`, указывающие, какую часть значения времени следует сохранять в базе данных.

Особенности Hibernate

По умолчанию Hibernate выбирает `TemporalType.TIMESTAMP`, если аннотация `@Temporal` отсутствует. Кроме того, мы отметили свойство Hibernate-аннотацией `@CreationTimestamp`. Она напоминает аннотацию `@Generated` из предыдущего раздела, вынуждая Hibernate автоматически генерировать значение поля. В этом случае Hibernate присвоит свойству значение текущего времени перед вставкой экземпляра сущности в базу данных. Еще одна похожая встроенная аннотация – `@UpdateTimestamp`. Кроме того, имеется возможность создавать и настраивать собственные генераторы значений, работающие в приложении или базе данных. Ознакомьтесь с аннотациями `org.hibernate.annotations.GeneratorType` и `ValueGenerationType`.

Другим особым типом свойств являются перечисления.

5.1.7. Отображение перечислений

Тип перечисления – это известная в языке Java идиома представления класса с постоянным (небольшим) количеством неизменяемых экземпляров. В приложении CaveatEmptor, например, можно было бы использовать следующее перечисление:

```
public enum AuctionType {
    HIGHEST_BID,
    LOWEST_BID,
    FIXED_PRICE
}
```

Теперь каждому товару (`Item`) можно присвоить свой тип `auctionType`:

```
@NotNull
@Enumerated(EnumType.STRING)   ← По умолчанию – ORDINAL
protected AuctionType auctionType = AuctionType.HIGHEST_BID;
```

В отсутствие аннотации `@Enumerated` Hibernate будет сохранять порядковый (`ORDINAL`) номер значения. Таким образом, он сохранит `1` для `HIGHEST_BID`, `2` для `LOWEST_BID` и `3` для `FIXED_PRICE`. Это довольно опасное поведение по умолчанию – если вы внесете изменения в перечисление `AuctionType`, существующие значения больше не будут отображаться на те же позиции. Следовательно, лучше выбрать вариант `EnumType.STRING` – Hibernate сохранит строковое представление значения перечисления.

На этом мы завершим обзор свойств основных типов и вариантов их отображения. К данному моменту мы рассмотрели типы свойств, предоставляемые JDK, такие как `String`, `Date` и `BigDecimal`. В вашей модели предметной области могут быть свои классы типов-значений, изображаемые на UML-диаграмме как композиции.

5.2. Отображение встраиваемых компонентов

До сих пор все отображаемые классы предметной модели были классами сущностей – каждый обладал собственной идентичностью и жизненным циклом. Тем не менее между классами `User` и `Address` установлен особый вид ассоциации, как показано на рис. 5.1.

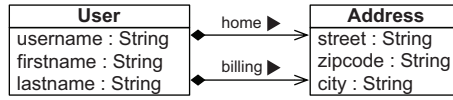


Рис. 5.1 ❖ Композиция классов `User` и `Address`

В терминах объектного моделирования эта ассоциация является разновидностью *агрегации* – отношением *часть целого*. Агрегация – это жесткая форма ассоциации, обладающая дополнительной семантикой, касающейся жизненного цикла объектов. В данном случае имеет место еще более жесткая форма – *композиция*, когда жизненный цикл части полностью зависит от жизненного цикла целого. Обычно класс, представляющий часть композиции в UML, такой как `Address`, становится потенциальным типом-значением в объектно-реляционном отображении.

5.2.1. Схема базы данных

При таком композиционном отношении мы отобразим класс `Address` как тип-значение, с той же семантикой, что и у классов `String` или `BigDecimlal`, а класс `User` отобразим как сущность. Сначала посмотрим на целевую SQL-схему, изображенную на рис. 5.2.

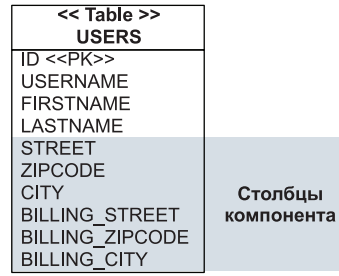


Рис. 5.2 ❖ Столбцы компонентов встроены в таблицу сущности

Здесь имеется лишь одна таблица, `USERS`, для отображения сущности `User`. В этой таблице хранятся все данные компонентов, и каждая запись представляет конкретного пользователя (`User`), а также его домашний адрес (`homeAddress`) и адрес оплаты (`billingAddress`). Если другая сущность будет ссылаться на адрес (`Address`) – например, `Shipment#deliveryAddress`, в таблице отправки (`SHIPMENT`) также появятся столбцы для хранения адреса (`Address`).

Такая схема отражает семантику типа-значения: конкретный адрес (`Address`) не может быть разделяемым, потому что у него отсутствует идентичность. Его первичным ключом является отображенный в базе данных идентификатор сущности-владельца. Встроенный компонент обладает зависимым жизненным циклом: при сохранении экземпляра сущности-владельца сохраняется экземпляр компонента. Экземпляр компонента удаляется при удалении экземпляра сущности-владельца. Hibernate не нужно даже выполнять специальный SQL-код – все данные находятся в одной записи.

Hibernate поддерживает хорошо детализированные модели предметных областей, когда количество классов превышает количество существующих таблиц. Давайте создадим классы и отображения для такого варианта.

5.2.2. Встраиваемые классы

В Java отсутствует понятие композиции – класс или свойство не может быть отмечено как компонент или иметь соответствующий жизненный цикл. Единственное отличие от сущности – это идентификатор в базе данных: у класса компонента нет собственной идентичности, и, следовательно, ему не требуется поле идентификатора или его отображение. Это обыкновенный POJO, как можно понять из листинга 5.6.

Листинг 5.6 ❖ Класс `Address` – встраиваемый компонент

Файл: /model/src/main/java/org/jpwh/model/simple/Address.java

```
@Embeddable  ← ❶ @Embeddable вместо аннотации @Entity
public class Address {

    @NotNull  ← Игнорируется при генерации DDL
    @Column(nullable = false)  ← Используется при генерации DDL
    protected String street;

    @NotNull
    @Column(nullable = false, length = 5)  ← Переопределяет VARCHAR(255)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String city;

    protected Address() {  ← ❷ Конструктор без аргументов
    }

    public Address(String street, String zipcode, String city) {  ← ❸ Вспомогательный
        this.street = street;                                        конструктор
        this.zipcode = zipcode;
        this.city = city;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
```

```
        this.street = street;
    }

    public String getZipcode() {
        return zipcode;
    }

    public void setZipcode(String zipcode) {
        this.zipcode = zipcode;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}
```

❶ Вместо `@Entity` этот компонент POJO отмечен аннотацией `@Embeddable`. Идентификатор отсутствует.
❷ Hibernate вызывает этот конструктор без аргументов для создания экземпляра и затем напрямую устанавливает значения полей.
❸ Для удобства можно создать дополнительные (общедоступные) конструкторы.

Все свойства встраиваемого класса по умолчанию сохраняются точно так же, как свойства хранимой сущности. Отображение свойств можно настраивать с помощью тех же аннотаций: `@Column` или `@Basic`. Свойства класса `Address` отображаются в столбцы `STREET`, `ZIPCODE` и `CITY` с ограничением `NOT NULL`.

Проблема: Hibernate Validator не генерирует ограничения NOT NULL

На момент написания книги в Hibernate Validator имеется нерешенная проблема: Hibernate не отображает аннотацию ограничения `@NotNull`, расположенную над свойствами встраиваемых компонентов, в ограничение `NOT NULL`, когда создает схему базы данных. Аннотация `@NotNull` над свойствами компонентов используется лишь для валидации с помощью Bean Validation во время выполнения. Чтобы сгенерировать ограничение в схеме, нужно отобразить свойство, используя аннотацию `@Column(nullable = false)`. В базе данных Hibernate эту проблему можно отслеживать по номеру `HVAL-3`.

Ниже приводится полное отображение. В определении сущности `User` нет ничего необычного:

Файл: /model/src/main/java/org/jpwh/model/simple/User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {
```

```
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    public Long getId() {
        return id;
    }

    protected Address homeAddress;

    public Address getHomeAddress() {
        return homeAddress;
    }

    public void setHomeAddress(Address homeAddress) {
        this.homeAddress = homeAddress;
    }

    // ...
}
```

Адрес (Address) является встраиваемым (@Embeddable) – аннотация не требуется

Обнаружив аннотацию `@Embeddable` перед классом `Address`, Hibernate отобразит столбцы `STREET`, `ZIPCODE` и `CITY` в таблицу сущности-владельца `USERS`. Обсуждая виды доступа к свойствам, мы отметили, что встраиваемые компоненты наследуют стратегию доступа от сущности-владельца. Это значит, что Hibernate будет обращаться к полям класса `Address`, используя ту же стратегию, что и для полей класса `User`. Это наследование также влияет на местоположение аннотаций отображения в классах встраиваемых компонентов:

- если сущность-владелец (`@Entity`) встроенного компонента отображается с использованием стратегии прямого доступа к полям, либо неявно, с помощью аннотации `@Id` перед полем, либо явно, с помощью аннотации `@Access(AccessType.FIELD)` перед классом, все аннотации отображения класса во встроенном компоненте должны располагаться над полями класса. То есть аннотации в классе `Address` должны находиться над полями, и Hibernate будет обращаться к ним во время выполнения напрямую. Наличие методов доступа в классе `Address` не обязательно;
- если сущность-владелец (`@Entity`) встроенного компонента отображается с использованием стратегии доступа к свойствам, либо неявно, с помощью аннотации `@Id` перед методом чтения, либо явно, с помощью аннотации `@Access(AccessType.PROPERTY)` перед классом, все аннотации отображения в классе встроенного компонента должны располагаться перед методами чтения. В этом случае Hibernate будет использовать методы доступа во встроенном компоненте;
- если встроенное свойство класса сущности-владельца – `User#homeAddress` в предыдущем примере – отметить аннотацией `@Access(AccessType.FIELD)`, Hibernate будет ожидать появления аннотаций перед полями класса `Address` и обращаться к ним во время выполнения напрямую;
- если встроенное свойство класса сущности-владельца – `User#homeAddress` в предыдущем примере – отметить аннотацией `@Access(AccessType.PROPER-`

TY), Hibernate будет ожидать появления аннотаций перед методами чтения в классе Address и обращаться к свойствам во время выполнения посредством методов доступа;

- если отметить аннотацией @Access сам встраиваемый класс, Hibernate будет использовать выбранную стратегию для чтения аннотаций отображения встраиваемого класса и соответствующий режим доступа к свойствам во время выполнения.

Особенности Hibernate

Важно помнить, что не существует способа изящно представить ссылку на экземпляр Address со значением null. Представьте, что бы получилось, если бы значения столбцов STREET, ZIPCOD и CITY могли отсутствовать. Что должен вернуть метод someUser.getHomeAddress(), когда Hibernate загружает пользователя (User) без адреса? В этом случае Hibernate вернет null. Кроме того, Hibernate сохраняет встроенные поля со значением null как NULL в столбцах, в которые отображается компонент. Следовательно, если сохранить пользователя (User) с «пустым» адресом (Address), когда экземпляр Address существует, но все его поля равны null, при загрузке пользователя (User) экземпляр Address возвращен не будет. Это может оказаться неожиданным, но, с другой стороны, вам в любом случае не следует использовать столбцы, в которых могут отсутствовать значения, а также следует избегать троичной логики.

Вам следует переопределить методы equals() и hashCode() класса Address и сравнивать экземпляры по значению. Это не особенно важно, пока не нужно сравнивать экземпляры: например, при добавлении их в HashSet. Мы будем обсуждать это позже, когда речь пойдет о коллекциях; см. раздел 7.2.1.

В более реалистичном сценарии у пользователя, возможно, будет несколько адресов для различных целей. На рис. 5.1 показано дополнительное отношение композиции между классами User и Address: поле billingAddress.

5.2.3. Переопределение встроенных атрибутов

В классе User имеется еще одно свойство-компонент – billingAddress. То есть в таблицу USERS должен сохраняться еще один экземпляр Address. Это создает конфликт отображения – на данный момент в схеме предусмотрены столбцы для хранения лишь одного экземпляра Address: STREET, ZIPCODE и CITY.

Для хранения еще одного экземпляра Address в каждой строке таблицы USERS понадобятся дополнительные столбцы. При отображении billingAddress придется переопределить имена столбцов.

Файл: /model/src/main/java/org/jpwh/model/simple/User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {

    @Embedded  ← Необязательная аннотация
```

```
    @AttributeOverrides({
            @AttributeOverride(name = "street",
                    column = @Column(name = "BILLING_STREET")),  ← Может быть NULL!
            @AttributeOverride(name = "zipcode",
                    column = @Column(name = "BILLING_ZIPCODE", length = 5)),
            @AttributeOverride(name = "city",
                    column = @Column(name = "BILLING_CITY"))
    })
    protected Address billingAddress;

    public Address getBillingAddress() {
        return billingAddress;
    }

    public void setBillingAddress(Address billingAddress) {
        this.billingAddress = billingAddress;
    }

    // ...
}
```

На самом деле аннотацию `@Embedded` можно опустить. Это альтернатива аннотации `@Embeddable` – достаточно отметить что-то одно: либо класс компонента, либо свойство класса-владельца (можно отметить и то, и другое, но это не даст никаких преимуществ). Аннотация `@Embedded` может пригодиться для отображения стороннего класса компонента, когда исходный код недоступен и нет возможности добавить аннотации, но используются правильные методы доступа (как в обычном классе JavaBeans).

Аннотация `@AttributeOverrides` выборочно переопределяет отображение свойств встроенного класса; в данном примере для всех трех свойств переопределяются имена столбцов. Теперь в таблице `USERS` можно хранить два экземпляра `Address` – каждому экземпляру соответствует свой набор столбцов (взгляните снова на схему на рис. 5.2).

Каждая аннотация `@AttributeOverride` перед свойством-компонентом является «конечной»: любые аннотации JPA или Hibernate над переопределяемым свойством будут игнорироваться. Это значит, что аннотации `@Column` в классе `Address` будут проигнорированы: все столбцы с префиксом `BILLING_*` смогут хранить значение `NULL`! (Однако Bean Validation по-прежнему будет распознавать аннотацию `@NotNull` перед свойством-компонентом; Hibernate лишь переопределит аннотации, связанные с хранением.)

Вы можете улучшить степень повторного использования предметной модели и сделать ее еще более детализированной, вкладывая встроенные компоненты друг в друга.

5.2.4. Отображение вложенных встраиваемых компонентов

Рассмотрим класс `Address` и то, как он инкапсулирует свои данные: вместо простой строки `city` можно было бы перенести данные во встраиваемый класс `City`. Посмотрите на модифицированную диаграмму предметной модели на рис. 5.3. Схема

SQL, которую мы будем использовать для отображения, содержит только одну таблицу USERS, как показано на рис. 5.4.

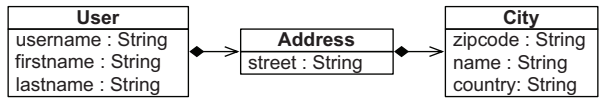


Рис. 5.3 ❖ Вложенная композиция классов Address и City

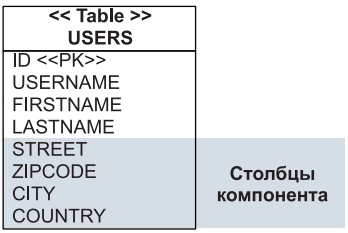


Рис. 5.4 ❖ Встроенные столбцы хранят данные из классов Address и City

Встраиваемый класс может иметь встроенное свойство. Класс Address теперь имеет свойство city:

Файл: /model/src/main/java/org/jpwh/model/advanced/Address.java

```
@Embeddable
public class Address {

    @NotNull
    @Column(nullable = false)
    protected String street;

    @NotNull
    @AttributeOverrides(
        @AttributeOverride(
            name = "name",
            column = @Column(name = "CITY", nullable = false)
        )
    )
    protected City city;

    // ...
}
```

Встраиваемый класс City имеет только свойства основных типов:

Файл: /model/src/main/java/org/jpwh/model/advanced/City.java

```
@Embeddable
public class City {
```

```
    @NotNull
    @Column(nullable = false, length = 5)  ← Переопределяет VARCHAR(255)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String name;

    @NotNull
    @Column(nullable = false)
    protected String country;

    // ...
}
```

Можно и дальше продолжить создание вложенных компонентов, выделив, например, класс `Country`. Независимо от того, насколько глубоко они находятся в композиции, все встроенные поля отображаются в столбцы таблицы сущности-владельца, в данном случае – таблицы `USERS`.

Вы можете размещать аннотации `@AttributeOverride` на любом уровне, как в случае со свойством `name` в классе `City`, когда оно отображается в столбец `CITY`. Этого можно добиться, либо (как показано) добавив аннотацию `@AttributeOverride` в класс `Address`, либо переопределив отображение в корневом классе сущности `User`. К вложенным полям можно обращаться через точку. Например, аннотация `@AttributeOveride(name = "city.name")` перед свойством `User#address` ссылается на атрибут `Address#City#name.`

Мы еще вернемся к встроенным компонентам в разделе 7.2. Вы можете также отображать коллекции компонентов или ссылаться из компонентов на сущности.

В начале этой главы мы говорили о свойствах основных типов и о том, как Hibernate отображает типы из JDK, такие как `java.lang.String`, в соответствующие типы SQL. Давайте узнаем больше об этой системе типов и преобразовании значений на более низком уровне.

5.3. Отображение типов Java и SQL с применением конвертеров

До этого момента предполагалось, что, когда отображается свойство типа `java.lang.String`, Hibernate сам выберет правильный SQL-тип. Но какое отображение между типами Java и SQL является корректным и как можно управлять выбором?

5.3.1. Встроенные типы

Каждая реализация JPA должна поддерживать минимальный набор преобразований типов между Java и SQL; вы видели этот список в начале текущей главы, в разделе 5.1. Hiberante поддерживает все эти отображения, а также некоторые нестандартные, но полезные на практике адаптеры. Сначала рассмотрим простые типы Java и их SQL-эквиваленты.

Простые и числовые типы

Встроенные типы, перечисленные в табл. 5.1, отображают примитивные типы Java и их обертки в соответствующие стандартные типы SQL. Мы также включили несколько других числовых типов.

Таблица 5.1. Простые типы Java, отображаемые в стандартные типы SQL

Имя	Тип Java	Тип ANSI SQL
integer	int, java.lang.Integer	INTEGER
long	long, java.lang.Long	BIGINT
short	short, java.lang.Short	SMALLINT
float	float, java.lang.Float	FLOAT
double	double, java.lang.Double	DOUBLE
byte	byte, java.lang.Byte	TINYINT
boolean	boolean, java.lang.Boolean	BOOLEAN
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC

Эти имена относятся только к Hibernate и потребуются далее для управления отображением типов.

Вы наверняка заметили, что ваша СУБД не поддерживает некоторых из упомянутых SQL-типов. Эти имена типов SQL являются именами стандартных типов ANSI. Большинство производителей СУБД игнорирует эту часть стандарта SQL, как правило, потому, что их устаревшая система типов возникла раньше стандарта. Но JDBC частично абстрагирует предоставляемые производителем типы данных, позволяя Hibernate работать со стандартными типами ANSI при выполнении таких DML-операций, как `INSERT` и `UPDATE`. При генерации схемы для конкретной системы Hibernate транслирует стандартные типы ANSI в соответствующие типы базы данных, используя настроенный диалект SQL. Это означает, что обычно нет причин беспокоиться о типах данных SQL, когда схема генерируется фреймворком Hibernate.

Если у вас уже есть готовая схема и/или вам нужно узнать, какие собственные типы данных имеются в СУБД, загляните в исходный код настроенного диалекта SQL. Например, диалект `H2Dialect`, поставляемый с Hibernate, содержит следующее отображение из ANSI-типа `NUMERIC` в тип базы данных `DECIMAL`: `registerColumnType(Types.NUMERIC, "decimal($p,$s)")`.

SQL-тип `NUMERIC` поддерживает настройку точности и масштаба числа. Точность и масштаб для поля типа `BigDecimal`, например, по умолчанию будут определены как `NUMERIC(19, 2)`. Чтобы изменить эти параметры при генерации схемы, поместите перед свойством аннотацию `@Column` и задайте значения параметров `precision` и `scale`.

Далее показаны типы, отображаемые в базу данных как строки.

Символьные типы

В табл. 5.2 показаны типы, отображающие символьные и строковые значения.

Таблица 5.2. Адаптеры для символьных и строковых значений

Имя	Тип java	Тип ANSI SQL
string	java.lang.String	VARCHAR
character	char[], Character[], java.lang.String	CHAR
yes_no	boolean, java.lang.Boolean	CHAR(1), 'Y' или 'N'
true_false	boolean, java.lang.Boolean	CHAR(1), 'T' или 'F'
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Система типов Hibernate выбирает тип данных SQL в зависимости от объявленной длины строкового значения: если поле типа `String` снабжено аннотацией `@Column(length = ...)` или `@Length` из Bean Validation, Hibernate выберет правильный тип данных SQL для заданной длины строки. На выбор также влияет настроенный диалект SQL. Например, для MySQL длина до 65 535 символов будет соответствовать столбцу обычного типа `VARCHAR(length)`, когда схема генерируется фреймворком Hibernate. Длина до 16 777 215 соответствует собственному типу MySQL – `MEDIUMTEXT`, а большие длины соответствуют типу `LONGTEXT`. Для всех свойств типа `java.lang.String` Hibernate устанавливает длину 255, поэтому без дополнительной настройки поле типа `String` будет отображаться в столбец `VARCHAR(255)`. Вы можете поменять логику выбора типов путем расширения класса вашего SQL-диалекта; читайте документацию с описанием диалекта и его исходный код, чтобы больше узнать о вашей СУБД.

Обычно база данных поддерживает интернационализацию текста, по умолчанию выбирая разумную (UTF-8) кодировку для всей базы данных или, по крайней мере, для целых таблиц. Эта настройка зависит от СУБД. Если вам потребуется явно управлять выбором кодировки или поменять типы столбцов на `NVARCHAR`, `NCHAR` или `NCLOB`, отметьте отображаемое свойство аннотацией `@org.hibernate.annotations.Nationalized`.

Для устаревших баз данных или СУБД с ограниченной системой типов, таких как Oracle, имеются встроенные конвертеры. СУБД Oracle не имеет даже логического типа – единственного типа данных, которого требует реляционная модель. Поэтому в большинстве существующих схем Oracle значения логического типа представлены как символы `Y/N` или `T/F`. Другим вариантом, который по умолчанию применяется диалектом Oracle в Hibernate, являются создание и использование столбца типа `NUMBER(1,0)`. И снова, если вам захочется узнать, как отображаются ANSI-типы в типы базы данных, обращайтесь к описанию диалекта SQL вашей СУБД.

Далее рассказывается о типах, которые в базе данных отображаются в типы даты и времени.

Типы даты и времени

В табл. 5.3 перечислены типы даты, времени и отметки времени.

Таблица 5.3. Типы даты и времени

Имя	Тип java	Тип ANSI SQL
date	java.util.Date, java.sql.Date	DATE
time	java.util.Date, java.sql.Time	TIME
timestamp	java.util.Date, java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
duration	java.time.Duration	BIGINT
instant	java.time.Instant	TIMESTAMP
localdatetime	java.time.LocalDateTime	TIMESTAMP
localdate	java.time.LocalDate	DATE
localtime	java.time.LocalTime	TIME
offsetdatetime	java.time.OffsetDateTime	TIMESTAMP
offsettime	java.time.OffsetTime	TIME
zoneddatetime	java.time.ZonedDateTime	TIMESTAMP

В своей предметной модели вы можете представлять дату и время при помощи `java.util.Date`, `java.util.Calendar` или подклассов `java.util.Date` из пакета `java.sql`. Это дело вкуса, и мы оставляем выбор за вами, но будьте последовательны. Возможно, вы не захотите привязывать классы предметной модели к типам из пакета JDBC.

Вы также можете использовать классы из пакета `java.time` в Java 8. Обратите внимание, что это относится только к Hibernate и не стандартизовано в JPA 2.1.

Поведение Hibernate, касающееся свойств типа `java.util.Date`, первое время может озадачивать: если сохранить свойство типа `java.util.Date`, Hibernate не вернет вам значение типа `java.util.Date` после загрузки из базы данных. Он вернет `java.sql.Date`, `java.sql.Time` или `java.sql.Timestamp`, в зависимости от того, какой параметр использовался в описании отображения свойства: `TemporalType.DATE`, `TemporalType.TIME` или `TemporalType.TIMESTAMP`.

При извлечении даты из базы данных Hibernate использует подкласс JDBC, потому что типы базы данных более точные, чем `java.util.Date`. Тип `java.util.Date` обеспечивает точность до миллисекунды, тогда как тип `java.sql.Timestamp` включает информацию о наносекундах, которые могут храниться в базе данных. Hibernate не будет отсекать эту информацию, чтобы вместить значение в `java.util.Date`. Такое поведение может привести к проблемам, если попытаться сравнить значения типа `java.util.Date`, используя метод `equals()` – он не симметричен с методом `equals()` подкласса `java.sql.Timestamp`.

Решается эта проблема просто, причем независимо от Hibernate, – не вызывайте `aDate.equals(bDate)`. Всегда сравнивайте значения времени и даты, используя значения времени Unix в миллисекундах (если точность до наносекунд не нуж-

на): выражение aDate.getTime() > bDate.getTime(), например, будет истинным (true) в том случае, когда aDate наступает позже bDate. Будьте бдительны: такие коллекции, как HashSet, тоже вызывают метод equals(). В подобных коллекциях не следует смешивать значения типов java.util.Date и java.sql.Date|Time|Timestamp. С полем типа Calendar такой проблемы не возникнет. Если вы сохранили значение типа Calendar, Hiberante всегда будет возвращать экземпляр Calendar, созданный при помощи метода Calendar.getInstance() (настоящий тип зависит от региональных настроек и часового пояса).

С другой стороны, можно написать свой *конвертер*, как будет показано далее в этой главе, и преобразовывать экземпляр любого типа java.sql.Date|Time|Timestamp, возвращаемый Hibernate, в обычный объект java.util.Date. Если при загрузке экземпляра Calendar из базы данных потребуется устанавливать часовой пояс, отличный от часового пояса по умолчанию, создание собственного конвертера может стать хорошей отправной точкой.

Далее представлены типы, сохраняемые в базе данных в двоичном виде.

Двоичные типы и типы для представления больших значений

В табл. 5.4 перечислены типы для работы с двоичными данными и большими значениями. Обратите внимание, что только тип binary может выступать в качестве типа свойств-идентификаторов.

Сначала нужно решить, как потенциально большое значение будет представлено в Hibernate: в двоичном виде или в виде текста.

Таблица 5.4. Двоичные типы и типы для представления больших значений

Имя	Тип java	Тип ANSI SQL
binary	byte[], java.lang.Byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	java.io.Serializable	VARBINARY

Если свойство хранимого класса имеет тип byte[], Hibernate отобразит его в столбец типа VARBINARY. Настоящий тип SQL будет зависеть от диалекта – например, в PostgresSQL это будет тип BYTEA, а в Oracle – RAW. В некоторых диалектах параметр length аннотации @Column также может повлиять на выбор типа в базе данных – например, в Oracle для длины 2000 и более будет выбран тип LONG RAW.

Свойство типа java.lang.String отображается в столбец типа SQL VARCHAR, и это поведение аналогично char[] и Character[]. Мы уже говорили, что некоторые диалекты выбирают разные типы, в зависимости от объявленной длины.

В обоих случаях Hibernate инициализирует значение свойства сразу же при загрузке экземпляра сущности, в котором есть это поле. Это может быть неудобно, когда приходится иметь дело с потенциально большими значениями, поэтому

обычно хочется переопределить отображение по умолчанию. В спецификации JPA для этой цели есть удобная аннотация `@Lob`:

```
@Entity
public class Item {

    @Lob
    protected byte[] image;

    @Lob
    protected String description;

    // ...
}
```

В данном случае тип `byte[]` будет отображаться в SQL-тип `BLOB`, а тип `String` – в `CLOB`. К сожалению, даже такая настройка не обеспечивает отложенную загрузку. Hibernate придется перехватывать доступ к полю и, например, загружать байты для поля `image` в момент вызова метода `someItem.getImage()`. Подобный подход требует внедрения дополнительного кода в байт-код классов после компиляции. Мы обсудим реализацию отложенной загрузки таким способом в разделе 12.1.3.

С другой стороны, можно поменять тип поля в Java-классе. JDBC напрямую поддерживает объекты логических указателей (LOB). Если для свойства в Java выбрать тип `java.sql.Clob` или `java.sql.Blob`, это обеспечит отложенную загрузку без внедрения в байт-код.

```
@Entity
public class Item {

    @Lob
    protected java.sql.Blob imageBlob;

    @Lob
    protected java.sql.Clob description;

    // ...
}
```

Эти классы JDBC поддерживают загрузку значений по требованию. В момент загрузки сущности владельца свойство получает значение указателя, а настоящие данные не загружаются сразу же. Если в рамках той же транзакции происходит обращение к свойству, данные загружаются или даже передаются напрямую (клиенту), не занимая временную память:

Файл: /examples/src/test/java/org/jpwh/test/advanced/LazyProperties.java

```
Item item = em.find(Item.class, ITEM_ID);

InputStream imageDataStream = item.getImageBlob().getBinaryStream();

ByteArrayOutputStream outStream = new ByteArrayOutputStream();
StreamUtils.copy(imageDataStream, outStream);
byte[] imageBytes = outStream.toByteArray();
```

Можно передавать поток байтов напрямую...

... или загружать их в память.

Недостаток заключается в том, что в этом случае предметная модель окажется связанной с JDBC; в модульных тестах обращение к свойствам типа LOB будет невозможно без подключения к базе данных.

Особенности Hibernate

В Hibernate имеются удобные методы для создания и установки значений типа `Blob` или `Clob`. В этом примере продемонстрированы чтение и пересылка `byteLength` байтов напрямую из потока `InputStream` в базу данных без сохранения во временной памяти:

```
Session session = em.unwrap(Session.class);   ← Нужен оригинальный Hibernate API
Blob blob = session.getLobHelper()
        .createBlob(imageInputStream, byteLength);   ← Необходимо знать количество байтов, которое нужно прочитать из потока
someItem.setImageBlob(blob);
em.persist(someItem);
```

Наконец, Hibernate предоставляет запасной механизм сериализации, работающий с полем любого типа, реализующего `java.io.Serializable`. Это отображение преобразует значение поля в поток байтов, сохраняемый в столбце типа `VARBINARY`. Сериализация и десериализация происходят при загрузке и сохранении сущности-владельца. Конечно, эту стратегию следует использовать с осторожностью, потому что данные будут существовать дольше, чем приложение. В один прекрасный день уже никто не сможет понять значения этих байтов в базе данных. Сериализация иногда полезна для хранения временных данных, таких как предпочтения пользователя, данные сеанса пользователя и т. д.

Hibernate выберет правильный вид адаптера в зависимости от Java-типа свойства. Если вам не нравится отображение по умолчанию, читайте дальше, чтобы узнать, как его переопределить.

Особенности Hibernate

Выбор типа адаптера

В предыдущих разделах вы уже видели многие адаптеры и их названия в Hibernate. Чтобы переопределить выбор типа по умолчанию, явно укажите имя конкретного адаптера:

```
@Entity
public class Item {

    @org.hibernate.annotations.Type(type = "yes_no")
    protected boolean verified = false;
}
```

Теперь вместо типа `BIT` это булево значение будет отображаться в столбец типа `CHAR` со значениями `Y` и `N`.

Также можно переопределить адаптер глобально в конфигурации загрузки Hibernate и указать собственный тип адаптера, о создании которого вы узнаете далее в этой главе:

```
metaBuilder.applyBasicType(new MyUserType(), new String[]{"date"});
```

Эта настройка переопределит встроенный тип адаптера с именем `date` и передаст управление преобразованиями полей типа `java.util.Date` вашей собственной реализации.

Мы рассматриваем такую расширяемую систему типов как одну из главных особенностей Hibernate и как главную причину его гибкости. Далее мы более подробно исследуем систему типов и пользовательские конвертеры JPA.

5.3.2. Создание собственных конвертеров JPA

Новым требованием к системе онлайн-аукциона является применение нескольких валют. Воплощение подобного требования может оказаться сложной задачей. Необходимо модифицировать схему базы данных, может понадобиться перенос данных из старой схемы в новую, а также придется обновить все приложения, имеющие доступ к базе данных. В этом разделе мы покажем, как JPA-конвертеры и расширяемая система типов Hibernate могут помочь в этом процессе, предоставляя гибкую дополнительную прослойку между приложением и базой данных.

Для работы с несколькими валютами мы добавим новый класс в предметную модель приложения CaveatEmptor – это `MonetaryAmount`, показанный в листинге 5.7.

Листинг 5.7 ❖ Неизменяемый класс типа-значения `MonetaryAmount`

Файл: /model/src/main/java/org/jpwh/model/advanced/MonetaryAmount.java

```
public class MonetaryAmount implements Serializable {

    protected final BigDecimal value;
    protected final Currency currency;

    public MonetaryAmount(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }

    public BigDecimal getValue() {
        return value;
    }

    public Currency getCurrency() {
        return currency;
    }

    public boolean equals(Object o) {

        if (this == o) return true;
```

❶ Класс типа-значения реализует интерфейс java.io.Serializable

❷ Специальный конструктор не нужен

❸ Реализация методов equals() и hashcode()

```
        if (!(o instanceof MonetaryAmount)) return false;

        final MonetaryAmount monetaryAmount = (MonetaryAmount) o;

        if (!value.equals(monetaryAmount.value)) return false;
        if (!currency.equals(monetaryAmount.currency)) return false;

        return true;
    }

    public int hashCode() {   ← ❸ Реализация методов equals() и hashcode()
        int result;
        result = value.hashCode();
        result = 29 * result + currency.hashCode();
        return result;
    }

    public String toString() {   ← ❹ Создает экземпляр MonetaryAmount из объекта String
        return getValue() + " " + getCurrency();
    }

    public static MonetaryAmount fromString(String s) {
        String[] split = s.split(" ");
        return new MonetaryAmount(
            new BigDecimal(split[0]),
            Currency.getInstance(split[1])
        );
    }
}
```

❶ Данный класс типа-значения должен реализовать интерфейс `java.io.Serializable`: когда Hibernate сохраняет данные экземпляра сущности в разделяемом кэше второго уровня (см. раздел 20.2), то он *разбирает* состояние сущности на составляющие. Если сущность имеет свойство типа `MonetaryAmount`, сериализованное представление значения поля сохраняется в кэше второго уровня. При загрузке данных сущности из кэша значение поля десериализуется и собирается обратно.

❷ Классу не требуется специальный конструктор. Его можно сделать неизменяемым даже при наличии полей с модификатором `final`, потому что ваш код – единственное место, где создаются его экземпляры.

❸ Вы должны реализовать методы `equals()` и `hashCode()` и сравнивать объекты, представляющие денежную сумму, «по значению».

❹ Вам понадобится строковое представление денежной суммы в виде экземпляра `String`. Реализуйте метод `toString()`, а также статический метод создания экземпляров из объектов `String`.

Далее следует обновить остальные части модели, задействовав класс `MonetaryAmount` для всех свойств, связанных с деньгами, таких как `Item#buyNowPrice` и `Bid#amount`.

Конвертация значений свойств основных типов

Как это обычно бывает, специалисты, отвечающие за базу данных, не могут реализовать поддержку нескольких валют прямо сейчас – им нужно больше времени.

Самое быстрое, что они могут сделать, – это изменить тип данных столбца. Они предложили хранить значение BUYNOWPRICE в таблице ITEM, в столбце типа VARCHAR, и добавлять код валюты в виде суффикса строкового представления денежной суммы. К примеру, хранимое значение может иметь вид 11.23 USD или 99 EUR.

При сохранении данных экземпляр MonetaryAmount необходимо преобразовать в объект String с подобным представлением. При загрузке данных объект String нужно преобразовать обратно в экземпляр MonetaryAmount.

Простейшее решение заключается в использовании javax.persistence.AttributeConverter – стандартного механизма расширения в JPA, – как показано в листинге 5.8.

Листинг 5.8 ❖ Преобразование строк в экземпляры MonetaryValue

Файл: /model/src/main/java/org/jpwh/converter/MonetaryAmountConverter.java

```
@Converter(autoApply = true)  ← Автоматически применяется к полям типа MonetaryAmount
public class MonetaryAmountConverter
    implements AttributeConverter<MonetaryAmount, String> {

    @Override
    public String convertToDatabaseColumn(MonetaryAmount monetaryAmount) {
        return monetaryAmount.toString();
    }

    @Override
    public MonetaryAmount convertToEntityAttribute(String s) {
        return MonetaryAmount.fromString(s);
    }
}
```

Конвертер должен реализовать интерфейс AttributeConverter; два аргумента определяют тип свойства Java и тип столбца в схеме базы данных соответственно. Java-тип в данном случае – это MonetaryAmount, а тип базы данных – String, отображаемый, как обычно, в SQL-тип VARCHAR. Класс должен быть отмечен аннотацией @Converter или объявлен конвертером в файле метаданных *orm.xml*. Если параметр autoApply включен, любое свойство типа MonetaryAmount в предметной модели (в классе-сущности или во встраиваемом классе) будет обрабатываться конвертером автоматически. (Пусть вас не вводит в заблуждение название метода convertToEntityAttribute() интерфейса AttributeConverter; имя выбрано не самое удачное.)

Примером свойства с типом MonetaryAmount в модели предметной области может послужить Item#buyNowPrice:

Файл: /model/src/main/java/org/jpwh/model/advanced/converter/Item.java

```
@Entity
public class Item {

    @NotNull
    @Convert(  ← Необязательная аннотация, т. к. активен параметр autoApply
```

```
        converter = MonetaryAmountConverter.class,
        disableConversion = false)
    @Column(name = "PRICE", length = 63)
    protected MonetaryAmount buyNowPrice;

    // ...
}
```

Аннотацию `@Convert` можно опустить: она применяется для переопределения или отключения конвертера для конкретного свойства. Аннотация `@Column` переименовывает столбец базы в `PRICE`; по умолчанию он получил бы имя `BUYNOWPRICE`. Для автоматического создания схемы его тип определяется как `VARCHAR` с длиной в 63 символа.

Позже, когда администраторы базы данных обновят схему и предоставят вам столбцы для хранения денежной суммы и валюты, вам придется внести изменения в нескольких местах в программе: удалить класс `MonetaryAmountConverter` из проекта и сделать класс `MonetaryAmount` встраиваемым с помощью аннотации `@Embeddable` – после этого он автоматически будет отображаться в два столбца базы данных. Также можно с легкостью выборочно подключать и отключать конвертеры, если некоторые таблицы в схеме еще не обновились.

Только что созданный конвертер работает с классом `MonetaryAmount` – новым классом в предметной модели. Но применение конвертеров не ограничивается пользовательскими классами – можно также переопределять встроенные адаптеры типов Hibernate. Например, можно создать собственный конвертер для некоторых или даже для всех свойств типа `java.util.Date` в предметной модели.

Конвертеры можно применять к свойствам классов сущностей, таких как `Item#buyNowPrice` из предыдущего примера. Но их также можно применять к свойствам встраиваемых классов.

Преобразование значений полей компонентов

В этой главе мы разрабатывали пример хорошо детализированной предметной модели. Ранее вы изолировали информацию об адресе пользователя (`User`) и отобразили встраиваемый класс `Address`. Давайте продолжим этот процесс и добавим наследование, используя абстрактный класс `Zipcode`, как показано на рис. 5.5.

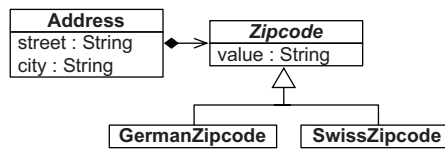


Рис. 5.5 ❖ Абстрактный класс `Zipcode` имеет два конкретных подкласса

Класс `Zipcode` тривиально прост, но не забудьте реализовать определение равенства по значению:

Файл: /model/src/main/java/org/jpwh/model/advanced/converter/Zipcode.java

```
abstract public class Zipcode {

    protected String value;

    public Zipcode(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Zipcode zipcode = (Zipcode) o;
        return value.equals(zipcode.value);
    }

    @Override
    public int hashCode() {
        return value.hashCode();
    }
}
```

Теперь можно инкапсулировать различие между немецкими и швейцарскими почтовыми индексами и их обработку в подклассах предметной модели:

Файл: /model/src/main/java/org/jpwh/model/advanced/converter/GermanZipcode.java

```
public class GermanZipcode extends Zipcode {

    public GermanZipcode(String value) {
        super(value);
    }
}
```

В подклассе пока не реализовано никакой особенной функциональности. Начнем с наиболее очевидного различия: немецкие почтовые индексы имеют длину в пять символов, швейцарские – четыре. Об этом позаботится наш специальный конвертер:

Файл: /model/src/main/java/org/jpwh/converter/ZipcodeConverter.java

```
@Converter
public class ZipcodeConverter
    implements AttributeConverter<Zipcode, String> {

    @Override
    public String convertToDatabaseColumn(Zipcode attribute) {
```

```
        return attribute.getValue();
    }

    @Override
    public Zipcode convertToEntityAttribute(String s) {
        if (s.length() == 5)
            return new GermanZipcode(s);
        else if (s.length() == 4)
            return new SwissZipcode(s);

        throw new IllegalArgumentException(
            "Unsupported zipcode in database: " + s
        );
    }
}
```

Если вы оказались тут, то подумайте о чистке базы данных... или создайте подкласс InvalidZipCode и возвращайте его в этом месте

При сохранении значения свойства Hibernate вызывает метод `convertToDatabaseColumn()` конвертера, который возвращает строковое представление в виде объекта `String`. Тип столбца в схеме – `VARCHAR`. При загрузке значения нужно проверить его длину и создать экземпляр класса `GermanZipcode` или `SwissZipcode`. Это ваша собственная процедура определения типа, и вам решать, какой тип Java выбрать для данного значения.

Теперь применим конвертер к какому-нибудь свойству типа `Zipcode` – например, к свойству `homeAddress` класса `User`:

Файл: /model/src/main/java/org/jpwh/model/advanced/converter/User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {

    @Convert(
        converter = ZipcodeConverter.class,
        attributeName = "zipcode"
    )
    protected Address homeAddress;

    // ...
}
```

Под одной аннотацией @Convert можно объединить преобразование нескольких атрибутов

Или «city.zipcode» для вложенных встраиваемых компонентов

Параметр `attributeName` объявляет атрибут `zipcode` встраиваемого класса `Address`. Можно использовать точечную нотацию для указания пути к атрибуту – если `zipcode` является не свойством класса `Address`, а свойством встраиваемого класса `City` (как было показано ранее в этой главе), путь к нему можно указать так: `city.zipcode`.

Если к одному встроенному свойству требуется применить несколько аннотаций `@Convert` – например, чтобы преобразовать несколько атрибутов класса `Address`, – их можно объединить в одну аннотацию `@Converts`. Конвертеры можно также применять к значениям коллекций и словарей, если их значения и/или ключи имеют основной или встраиваемый тип. К примеру, можно добавить анно-

тацию @Convert перед хранимым свойством типа Set<Zipcode>. Позже, в главе 7, мы покажем, как отображать хранимые коллекции при помощи аннотации @ElementCollection.

Для хранимых словарей параметр attributeName аннотации @Convert имеет специальный синтаксис:

- если свойство zipcode имеет тип Map<Address, String>, применить конвертер к каждому ключу в словаре можно при помощи атрибута key.zipcode;
- если свойство zipcode имеет тип Map<String, Address>, применить конвертер к каждому значению в словаре можно при помощи аттрибута value.zipcode;
- если свойство zipcode имеет тип Map<Zipcode, String>, применить конвертер к ключу каждой записи в словаре можно при помощи атрибута key;
- если свойство zipcode имеет тип Map<String, Zipcode>, применить конвертер к значению каждой записи в словаре можно при помощи атрибута attributeName.

Как и прежде, если встраиваемые классы вложены, в качестве имен атрибутов могут указываться пути, записанные в точечной нотации; можно написать key.city.zipcode, чтобы сослаться на свойство zipcode класса City в композиции с классом Address.

Конвертеры JPA имеют следующие ограничения:

- их нельзя применять к свойству-идентификатору или свойству с версией сущности;
- не следует применять конвертеры к свойствам, отмеченным аннотациями @Enumerated или @Temporal, потому что эти аннотации уже определяют необходимые преобразования. Если потребуется применить собственный конвертер к перечислениям или свойствам с датой/временем, не отмечайте их аннотацией @Enumerated или @Temporal;
- конвертер можно применить к свойству, отображение которого описывается в файле *hbm.xml*, но тогда к имени следует добавить префикс: type="converter:qualified.ConverterName".

Вернемся к поддержке нескольких валют в приложении CaveatEmptor. Администраторы базы данных в очередной раз обновили схему и попросили вас обновить приложение.

Особенности Hibernate

5.3.3. Расширение Hibernate с помощью пользовательских типов

Наконец мы добавили в схему базы данных новые столбцы для поддержки нескольких валют. Теперь в таблице ITEM есть столбец BUYNOWPRICE_AMOUNT и отдельный столбец BUYNOWPRICE_CURRENCY, представляющий валюту. Также имеются два столбца INITIALPRICE_AMOUNT и INITIALPRICE_CURRENCY. Эти столбцы требуется отобразить в свойства buyNowPrice и initialPrice типа MonetaryAmount в классе Item.

В идеале хотелось бы избежать необходимости изменять предметную модель; свойства уже используют класс `MonetaryAmount`. К сожалению, стандартизированные конвертеры JPA не поддерживают преобразования значений в несколько или из нескольких столбцов. Другое ограничение конвертеров JPA – интеграция с механизмом запросов. Невозможно написать следующий запрос: `select i from Item i where i.buyNowPrice.amount > 100`. Благодаря конвертеру из предыдущего раздела Hibernate знает, как преобразовать класс `MonetaryAmount` в строку и обратно. Но ему неизвестно о существовании атрибута `amount` в классе `MonetaryAmount`, поэтому он не сможет разобрать подобного запроса.

Эту проблему легко решить, отобразив класс `MonetaryAmount` как встраиваемый (`@Embeddable`), подобно классу `Address`, который мы обсудили ранее в этой главе. Каждое поле класса `MonetaryAmount` – `amount` и `currency` – будет отображаться в соответствующий столбец базы данных.

Однако администраторы базы данных добавили к требованиям поправку: поскольку другое, более старое приложение также обращается к базе данных, необходимо преобразовывать каждую сумму в целевую валюту перед сохранением в базу данных. К примеру, значение `Item#buyNowPrice` должно храниться в долларах США, а значение `Item#initialPrice` должно храниться в евро. (Если этот пример показался вам притянутым за уши, можем вас уверить, что в реальном мире встречаются куда худшие примеры. Эволюция используемой совместно схемы базы данных может оказаться затратной, но она тем не менее необходима, потому что данные всегда существуют дольше приложений.) Hibernate имеет собственный API для конвертера – точку расширения для доступа на более низком уровне.

Точки расширения

Интерфейсы для расширения системы типов Hibernate определены в пакете `org.hibernate.usertype` и включают:

- `UserType` – позволяет преобразовывать значения посредством взаимодействий с обычными классами JDBC: `PreparedStatement` при сохранении и `ResultSet` при загрузке. Реализовав этот интерфейс, вы сможете контролировать порядок кэширования значений и проверки изменений состояний объектов в Hibernate. Адаптер для класса `MonetaryAmount` должен реализовать этот интерфейс;
- `CompositeUserType` – расширяет `UserType`, предоставляя фреймворку Hibernate больше подробностей об адаптируемом классе. Вы можете указать, что компонент типа `MonetaryAmount` имеет два поля: `amount` и `currency`, – и затем обращаться к этим полям в запросах, используя точечную нотацию, например: `select avg(i.buyNowPrice.amount) from Item i`;
- `ParameterizedUserType` – добавляет настройки отображений в класс адаптера. Вы должны реализовать этот интерфейс для преобразования объектов `MonetaryAmount`, потому что в некоторых отображениях нужно конвертировать сумму в доллары США, а в других – в евро. Вам придется написать всего один адаптер, а затем вы сможете настраивать его поведение для отображения полей;

- `DynamicParameterizedType` – более мощный API управления настройками, открывающий доступ к динамической информации адаптера, такой как имена таблиц и столбцов, в которые выполняется отображение. Вы можете запросто выбрать его вместо `ParameterizedUserType` – это не повлечет дополнительных затрат и не увеличит сложности;
- `EnhancedUserType` – необязательный интерфейс для адаптеров свойств-идентификаторов и селекторов. В отличие от конвертеров JPA, класс `UserType` в Hibernate может служить адаптером свойства сущности любого типа. Но поскольку класс `MonetaryAmount` не является типом свойства-идентификатора или селектора, вам этот интерфейс не понадобится;
- `UserVersionType` – необязательный интерфейс для адаптеров свойств, хранящих номера версий объектов;
- `UserCollectionType` – этот редко используемый интерфейс применяется для реализации собственных коллекций. Его нужно реализовать при работе с коллекциями, имеющими типы не из JDK, а также для сохранения дополнительной семантики.

В адаптере для класса `MonetaryAmount` мы реализуем некоторые из этих интерфейсов.

Реализация интерфейса UserType

Класс `MonetaryAmountUserType` довольно обширный, поэтому мы будем изучать его в несколько этапов. Ниже перечислены интерфейсы, которые он реализует:

Файл: /model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```
public class MonetaryAmountUserType
    implements CompositeUserType, DynamicParameterizedType {

    // ...
}
```

Сначала реализуем интерфейс `DynamicParameterizedType`. Выбор целевой валюты для конвертации производится путем анализа параметра отображения:

Файл: /model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```
protected Currency convertTo;

public void setParameterValues(Properties parameters) {

    ParameterType parameterType =  ← ❶ Обращение к динамическим параметрам
        (ParameterType) parameters.get(PARAMETER_TYPE);
    String[] columns = parameterType.getColumns();
    String table = parameterType.getTable();
    Annotation[] annotations = parameterType.getAnnotationsMethod();

    String convertToParameter = parameters.getProperty("convertTo");
    this.convertTo = Currency.getInstance(
        convertToParameter != null ? convertToParameter : "USD"
    );
}
```

❷ Определение целевой валюты

❶ Здесь можно получить доступ к некоторым динамическим параметрам, таким как имена столбцов, в которые производится отображение, таблицы (сущности) или даже аннотации поля или метода чтения отображаемого свойства. Хотя в этом примере они и не потребуются.
❷ Для определения целевой валюты при сохранении значения в базу данных потребуется только параметр `convertTo`. Если параметр не установлен, по умолчанию выбираются доллары США.

Далее следует вспомогательный код, реализующий интерфейс `UserType`:

Файл: /model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```
public Class returnedClass() {  ← ❶ Адаптируемый класс
    return MonetaryAmount.class;
}

public boolean isMutable() {  ← ❷ Позволяет включить оптимизацию
    return false;
}

public Object deepCopy(Object value) {  ← ❸ Копирует значение
    return value;
}

public Serializable disassemble(Object value,  ← ❹ Возвращает сериализованное представление
                                SessionImplementor session) {
    return value.toString();
}

public Object assemble(Serializable cached,  ← ❺ Создает экземпляр MonetaryAmount
                       SessionImplementor session, Object owner) {
    return MonetaryAmount.fromString((String) cached);
}

public Object replace(Object original, Object target,  ← ❻ Возвращает копию оригинала
                      SessionImplementor session, Object owner) {
    return original;
}

public boolean equals(Object x, Object y) {  ← ❼ Определяет, поменялось ли значение
    return x == y || !(x == null || y == null) && x.equals(y);
}

public int hashCode(Object x) {
    return x.hashCode();
}
```

❶ Метод `returnedClass` возвращает адаптируемый класс – в данном случае `MonetaryAmount`.
❷ Если известно, что класс `MonetaryAmount` неизменяемый, Hibernate может применять некоторые оптимизации.
❸ Если фреймворку Hibernate потребуется скопировать значение, он вызовет этот метод. В случае неизменяемых классов, таких как `MonetaryAmount`, можно возвращать переданное значение.

❹ Hiberante вызывает `disassemble`, когда сохраняет значение в глобальном разделяемом кэше второго уровня. Метод должен вернуть сериализованное значение в виде экземпляра `Serializable`. В случае с классом `MonetaryAmount` проще всего вернуть представление в виде объекта `String`. Или, поскольку класс `MonetaryAmount` сам реализует интерфейс `Serializable`, можно вернуть его экземпляр непосредственно.

❺ Hibernate вызывает этот метод, когда читает сериализованное представление из глобального разделяемого кэша второго уровня. Экземпляр `MonetaryAmount` создается из строкового представления в виде объекта `String`. Если бы в кэше хранился сериализованный экземпляр `MonetaryAmount`, можно было бы вернуть его непосредственно.

❻ Этот метод вызывается во время операций слияния, выполняемых методом `EntityManager#merge()`. Он должен вернуть копию оригинала. Или, если тип-значение является неизменяемым, как `MonetaryAmount`, можно вернуть оригинал.

❼ Чтобы определить факт изменения и необходимость записи в базу данных, Hibernate использует проверку на равенство по значению. Можно положиться на процедуру определения равенства, уже написанную для класса `MonetaryAmount`.

Настоящую работу адаптер выполняет в моменты загрузки и сохранения значений, как показано в реализациях следующих методов:

Файл: /model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```
public Object nullSafeGet(ResultSet resultSet,  ← ❶ Читает экземпляр ResultSet
                          String[] names,
                          SessionImplementor session,
                          Object owner) throws SQLException {

    BigDecimal amount = resultSet.getBigDecimal(names[0]);
    if (resultSet.wasNull())
        return null;
    Currency currency =
        Currency.getInstance(resultSet.getString(names[1]));
    return new MonetaryAmount(amount, currency);
}

public void nullSafeSet(PreparedStatement statement,  ← ❷ Сохраняет экземпляр
                        Object value,
                        int index,
                        SessionImplementor session) throws SQLException {

    if (value == null) {
        statement.setNull(
            index,
            StandardBasicTypes.BIG_DECIMAL.sqlType());
        statement.setNull(
            index + 1,
            StandardBasicTypes.CURRENCY.sqlType());
    } else {
        MonetaryAmount amount = (MonetaryAmount) value;
        MonetaryAmount dbAmount = convert(amount, convertTo);  ← Конвертировать в целевую валюту при сохранении MonetaryAmount
        statement.setBigDecimal(index, dbAmount.getValue());
```

```
            statement.setString(index + 1, convertTo.getCurrencyCode());
        }
    }

    protected MonetaryAmount convert(MonetaryAmount amount,   ← ❸ Конвертирует валюту
                                     Currency toCurrency) {
        return new MonetaryAmount(
            amount.getValue().multiply(new BigDecimal(2)),
            toCurrency
        );
    }
```

❶ Этот метод читает объект `ResultSet`, когда экземпляр `MonetaryAmount` извлекается из базы данных. Метод получает значения `amount` и `currency` из результата запроса и создает экземпляр класса `MonetaryAmount`.

❷ Этот метод вызывается, когда требуется сохранить объект `MonetaryAmount` в базу данных. Сумма пересчитывается в целевую валюту, и затем значения `amount` и `currency` сохраняются в экземпляре `PreparedStatement` (если только экземпляр `MonetaryAmount` не `null` – в этом случае для подготовки выражения вызывается `setNull()`).

❸ Здесь можно реализовать любые правила преобразования. Для целей примера значение просто удваивается, чтобы можно было легко убедиться, что преобразование прошло успешно. Этот код нужно поменять на настоящую конвертацию в реальном приложении. Данный метод не является частью интерфейса `UserType`.

Далее следуют методы интерфейса `CompositeUserType`, помогающие получить сведения о классе `MonetaryAmount`, чтобы Hibernate мог интегрировать этот класс в механизм запросов:

Файл: /model/src/main/java/org/jpwh/converter/MonetaryAmountUserType.java

```
public String[] getPropertyNames() {
    return new String[]{"value", "currency"};
}

public Type[] getPropertyTypes() {
    return new Type[]{
        StandardBasicTypes.BIG_DECIMAL,
        StandardBasicTypes.CURRENCY
    };
}

public Object getPropertyValue(Object component,
                               int property) {
    MonetaryAmount monetaryAmount = (MonetaryAmount) component;
    if (property == 0)
        return monetaryAmount.getValue();
    else
        return monetaryAmount.getCurrency();
}

public void setPropertyValue(Object component,
```

```
                              int property,
                              Object value) {
    throw new UnsupportedOperationException(
        "MonetaryAmount is immutable"
    );
}
```

Работа над классом `MonetaryAmountUserType` завершена, и теперь его можно использовать в `@org.hibernate.annotations.Type`, как показано в разделе «Выбор адаптера типа» выше. Эта аннотация также поддерживает параметры и позволяет указать целевую валюту с помощью параметра `convertTo`.

Мы советуем создавать *определения типов*, связывая адаптеры с некоторыми параметрами.

Использование определений типов

Итак, нам нужен один адаптер для преобразования в доллары США и еще один для преобразования в евро. Если один раз объявить эти параметры как *определение типа*, вам не придется повторять их в отображениях свойств. Лучшим местом для объявления определений типов является файл метаданных уровня пакета *package-info.java*:

Файл: /model/src/main/java/org/jpwh/converter/package-info.java

```
@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name = "monetary_amount_usd",
        typeClass = MonetaryAmountUserType.class,
        parameters = {@Parameter(name = "convertTo", value = "USD")}
    ),
    @org.hibernate.annotations.TypeDef(
        name = "monetary_amount_eur",
        typeClass = MonetaryAmountUserType.class,
        parameters = {@Parameter(name = "convertTo", value = "EUR")}
    )
})
package org.jpwh.converter;
import org.hibernate.annotations.Parameter;
```

Теперь можно использовать адаптеры в отображениях, используя названия `monetary_amount_usd` и `monetary_amount_eur`.

Давайте отобразим свойства `buyNowPrice` и `initialPrice` класса `Item`:

Файл: /model/src/main/java/org/jpwh/model/advanced/usertype/Item.java

```
@Entity
public class Item {

    @NotNull
    @org.hibernate.annotations.Type(
        type = "monetary_amount_usd"
```

```
    )
    @org.hibernate.annotations.Columns(columns = {
        @Column(name = "BUYNOWPRICE_AMOUNT"),
        @Column(name = "BUYNOWPRICE_CURRENCY", length = 3)
    })
    protected MonetaryAmount buyNowPrice;

    @NotNull
    @org.hibernate.annotations.Type(
        type = "monetary_amount_eur"
    )
    @org.hibernate.annotations.Columns(columns = {
        @Column(name = "INITIALPRICE_AMOUNT"),
        @Column(name = "INITIALPRICE_CURRENCY", length = 3)
    })
    protected MonetaryAmount initialPrice;

    // ...
}
```

Класс `UserType` преобразует значения только для одного столбца, поэтому аннотация `@Column` не требуется. Однако классу `MonetaryAmountUserType` нужен доступ к двум столбцам, поэтому в отображении поля нужно явно объявить два столбца. Но поскольку JPA не поддерживает нескольких аннотаций `@Column` перед одним свойством, их следует сгруппировать, используя аннотацию `@org.hibernate.annotations.Columns`. Обратите внимание, что порядок аннотаций теперь имеет значение! Посмотрите еще раз на код класса `MonetaryAmountUserType` – многие операции зависят от обращения к массиву по индексу. Элементы в `PreparedStatement` или `ResultSet` следуют в том же порядке, что и столбцы в отображении. Заметьте также, что количество столбцов не играет роли при выборе между `UserType` и `CompositeUserType` – только желание передавать механизму запросов свойства с типами-значениями.

Используя класс `MonetaryAmountUserType`, вы расширили прослойку между предметной моделью в Java и схемой базы данных SQL. Оба представления теперь более устойчивы к изменениям, и вы можете справиться с еще более необычными требованиями, не внося изменений в классы предметной модели.

5.4. Резюме

- Мы обсудили отображение свойств классов сущностей с основными и встраиваемыми типами.
- Узнали, как переопределять основные отображения, как менять имена столбцов, в которые производится отображение, как использовать вычисляемые свойства, а также свойства со значениями по умолчанию, хранящие дату и/или время и перечисления.
- Исследовали классы встраиваемых компонентов и процесс создания хорошо детализированных моделей предметной области.

- Научились отображать поля нескольких Java-классов, участвующих в композиции, таких как `Address` и `City`, в одну таблицу сущности.
- Узнали, как Hibernate выбирает конвертер между типами Java и SQL, а также какие типы поддерживаются фреймворком Hibernate по умолчанию.
- Создали собственный конвертер для класса `MonetaryAmount`, используя стандартные интерфейсы расширения JPA, а затем и низкоуровневый адаптер, наследующий низкоуровневый интерфейс `UserType`.

Глава 6

Отображение наследования

В этой главе:
- стратегии отображения наследования;
- полиморфные ассоциации.

До сих пор мы намеренно почти ничего не говорили о наследовании. Отображение иерархии классов в таблицы может оказаться весьма сложной задачей, и в этой главе мы представим различные стратегии ее решения.

Основной стратегией отображения классов в таблицы базы данных может быть «одна таблица для каждого класса хранимой сущности». Такой подход выглядит простым и действительно отлично работает, но только пока не приходится иметь дела с наследованием.

Наследование представляет собой такое наглядное структурное несоответствие между объектно-ориентированным и реляционным мирами, потому что объектно-ориентированные системы моделируют оба вида отношений: *является* (*is a*, то есть один класс является специализацией другого) и *имеет* (*has a*, в классе определена ссылка на экземпляр другого класса). Модели, основанные на SQL, поддерживают только отношение *имеет* (*has a*); СУБД SQL не поддерживают наследования типов, и даже когда такая поддержка имеется, она является либо нестандартной, либо неполной.

Существуют четыре стратегии представления иерархии наследования:
- одна таблица для каждого конкретного класса и полиморфное поведение по умолчанию во время выполнения;
- одна таблица для каждого конкретного класса, но полное исключение полиморфизма и отношений наследования из схемы SQL. Для полиморфного поведения во время выполнения необходимо использовать запросы с SQL-инструкцией `UNION`;
- единая таблица для целой иерархии классов: полиморфизм поддерживается за счет денормализации схемы SQL, а определение суперкласса/подкласса осуществляется посредством различения строк;

- одна таблица для каждого подкласса: отношение *is a* (наследования) представлено в виде отношения *has a* (связь по внешнему ключу) с применением SQL-операции `JOIN`.

В этой главе показан нисходящий подход к проектированию, предполагающий, что вы начинаете с модели предметной области и постепенно пытаетесь получить новую схему SQL. Описанные стратегии отображения могут применяться и в том случае, когда вы работаете снизу вверх, начиная с существующих таблиц в базе данных. По ходу изложения мы продемонстрируем пару приемов, позволяющих работать с несовершенными схемами таблиц.

6.1. Одна таблица для каждого конкретного класса и неявный полиморфизм

Предположим, что мы выбрали простейший из предложенных вариантов: в точности одна таблица для каждого конкретного класса. Отобразить все свойства класса, в том числе унаследованные, в столбцы этой таблицы можно, как показано на рис. 6.1.

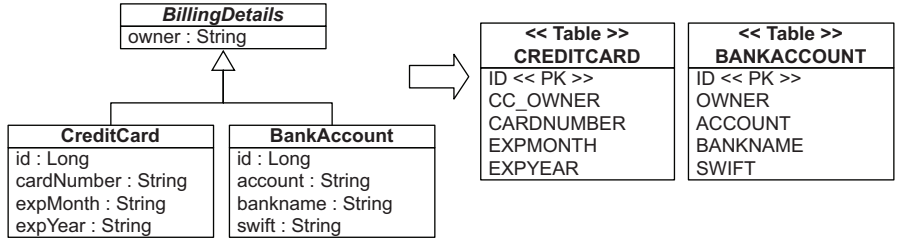


Рис. 6.1 ❖ Отображение каждого конкретного класса в независимую таблицу

Опираясь на такой неявный полиморфизм, каждый конкретный класс можно отобразить как обычно, при помощи аннотации `@Entity`. Свойства суперкласса по умолчанию игнорируются и не сохраняются! Чтобы встроить свойства суперкласса в таблицы конкретных подклассов, необходимо отметить их аннотацией `@MappedSuperclass`; см. листинг 6.1.

Листинг 6.1 ❖ Отображение класса `BillingDetails` (абстрактного суперкласса) с неявным полиморфизмом

Файл: /model/src/main/java/org/jpwh/model/inheritance/mappedsuperclass/BillingDetails.java

```
@MappedSuperclass
public abstract class BillingDetails {

    @NotNull
    protected String owner;

    // ...
}
```

Теперь отобразим конкретные подклассы.

Листинг 6.2 ❖ Отображение класса `CreditCard` (конкретный подкласс)

Файл: /model/src/main/java/org/jpwh/model/inheritance/mappedsuperclass/CreditCard.java

```
@Entity
@AttributeOverride(
        name = "owner",
        column = @Column(name = "CC_OWNER", nullable = false))
public class CreditCard extends BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;

    // ...
}
```

Отображение класса `BankAccount` выглядит аналогично, поэтому мы его опустим.

В подклассе можно переопределить отображение одного или нескольких столбцов суперкласса, используя аннотации `@AttributeOverride` или `@AttributeOverrides` соответственно. В предыдущем примере столбец `OWNER` в таблице `CREDITCARD` был переименован в `CC_OWNER`.

Также в суперклассе можно объявить свойство идентификатора, используя общие для всех подклассов имя столбца и стратегию, чтобы не повторять их объявления. Мы не сделали этого в примере, показывая, что это необязательно.

Главная проблема неявного отображения наследования заключается в отсутствии достаточной поддержки полиморфных ассоциаций. Обычно в базе данных ассоциации представлены в виде связей по внешнему ключу. Если все подклассы отображаются на разные таблицы, как показано в схеме на рис. 6.1, полиморфные ассоциации с их суперклассом (абстрактным классом `BillingDetails`) не могут быть представлены в виде простой связи по внешнему ключу. У вас не может быть другой сущности, отображаемой с внешним ключом, «ссылающимся на таблицу `BILLINGDETAILS`», – такой таблицы попросту нет. А это создаст проблемы для предметной модели, поскольку класс `BillingDetails` связан с классом `User` отношением ассоциации; обеим таблицам – `CREDITCARD` и `BANKACCOUNT` – понадобится внешний ключ, ссылающийся на таблицу `USERS`. Ни одна из этих проблем не имеет простого решения, поэтому придется рассмотреть альтернативную стратегию отображения.

Проблему также представляют полиморфные запросы, возвращающие экземпляры всех классов, реализующих запрашиваемый интерфейс. Hibernate должен выполнять запрос для суперкласса в виде нескольких SQL-выражений `SELECT` – по одному для каждого подкласса. JPA-запрос `select bd from BillingDetails bd` требует выполнения двух выражений SQL:

```
select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT
from
    BANKACCOUNT
select
    ID, CC_OWNER, CARDNUMBER, EXPMONTH, EXPYEAR
from
    CREDITCARD
```

Hibernate использует отдельный SQL-запрос для каждого конкретного подкласса. С другой стороны, запросы для конкретных классов просты и обладают хорошей производительностью: Hibernate выполнит только один запрос.

Другой концептуальной проблемой данной стратегии отображения является разделение одинаковой семантики несколькими разными столбцами в разных таблицах. Это сильно усложняет эволюцию схемы. Например, переименование или изменение типа свойства суперкласса вызовет изменение столбцов во многих таблицах. Большинство стандартных операций рефакторинга, предоставляемых IDE, потребует ручной доработки, поскольку автоматизированные процедуры, как правило, не учитывают таких особенностей, как `@AttributeOverrides`. Это также усложнит реализацию ограничений целостности базы данных, применяемых ко всем подклассам.

Мы советуем применять этот подход (только) для верхушки иерархии классов, где полиморфизм на самом деле не нужен и не предвидится изменений в суперклассе в будущем. Но это не подходит для предметной модели приложения *CaveatEmptor*, где запросы и прочие сущности ссылаются на класс `BillingDetails`.

Большинство проблем с полиморфными запросами и ассоциациями решается при помощи SQL-выражения `UNION`.

6.2. Одна таблица для каждого конкретного класса с объединениями

Для начала рассмотрим отображение подкласса с объединением, где в качестве абстрактного класса (или интерфейса) выступает класс `BillingDetails`, как и в предыдущем разделе. В этом случае снова имеются две таблицы с повторяющимися столбцами, соответствующими свойствам суперкласса в обеих: `CREDITCARD` и `BANKACCOUNT`. Новой является стратегия наследования под названием `TABLE_PER_CLASS`, объявленная в суперклассе, как показано далее.

Листинг 6.3 ❖ Отображение класса `BillingDetails` с применением стратегии `TABLE_PER_CLASS`

Файл: /model/src/main/java/org/jpwh/model/inheritance/tableperclass/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String owner;

    // ...
}
```

В суперклассе должны иметься идентификатор в базе данных и его отображение, чтобы все подклассы могли использовать его в своих таблицах. В отличие от предыдущей стратегии, это условие обязательно. Обе таблицы – `CREDITCARD` и `BANKACCOUNT` – имеют столбец первичного ключа `ID`. Все отображения конкретных классов наследуют хранимые свойства суперкласса (или интерфейса). Требуется только добавить аннотацию `@Entity` перед каждым подклассом.

Листинг 6.4 ❖ Отображение класса `CreditCard`

Файл: /model/src/main/java/org/jpwh/model/inheritance/tableperclass/CreditCard.java

```
@Entity
public class CreditCard extends BillingDetails {

    @NotNull
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;

    // ...
}
```

Помните, что схема SQL по-прежнему ничего не знает о наследовании – таблицы выглядят в точности, как показано на рис. 6.1.

Обратите внимание, что согласно стандарту JPA стратегия `TABLE_PER_CLASS` не является обязательной, поэтому она может поддерживаться не всеми реализациями JPA. Реализация также зависит от конкретного воплощения JPA – в Hibernate она является эквивалентом отображения `<union-subclass>` в старом оригинальном XML-файле метаданных Hibernate (не беспокойтесь об этом, если вам никогда не приходилось использовать оригинальные XML-файлы Hibernate).

Если бы класс BillingDetails был конкретным, понадобилась бы дополнительная таблица для хранения экземпляров. Еще раз подчеркнем, что между таблицами в базе данных по-прежнему нет никаких отношений, кроме наличия некоторых (многих) одинаковых столбцов.

Преимущества данной стратегии отображения станут очевидны после знакомства с полиморфными запросами. К примеру, запрос select bd from BillingDetails bd генерирует следующее выражение SQL:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, CLAZZ_
 from
    ( select
          ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
          null as ACCOUNT,
          null as BANKNAME,
          null as SWIFT,
          1 as CLAZZ_
    from
        CREDITCARD
    union all
    select
        id, OWNER,
        null as EXPMONTH,
        null as EXPYEAR,
        null as CARDNUMBER,
        ACCOUNT, BANKNAME, SWIFT,
        2 as CLAZZ_
    from
        BANKACCOUNT
  ) as BILLINGDETAILS
```

Этот запрос SELECT использует подзапрос в предложении FROM для извлечения всех экземпляров класса BillingDetails из всех таблиц конкретных классов. Таблицы объединяются с помощью оператора UNION, а в промежуточный результат вставляются литералы (в данном случае это 1 и 2); они нужны фреймворку Hibernate для создания экземпляра правильного класса из данных в конкретной записи. Объединение требует, чтобы участвующие в нем запросы имели одинаковую структуру столбцов, поэтому вместо несуществующих столбцов пришлось вставить NULL. Вы спросите: действительно ли этот запрос выполнится быстрее, чем два отдельных выражения? Здесь можно предоставить оптимизатору базы данных найти лучший план выполнения для объединения строк из нескольких таблиц вместо слияния результатов двух запросов в памяти, как это сделал бы механизм полиморфной загрузки в Hibernate.

Другим гораздо более важным преимуществом является возможность применения полиморфных ассоциаций; например, теперь стало возможным отобразить ассоциацию от класса User к классу BillingDetails. Hibernate может использовать

запрос с выражением UNION для имитации единой таблицы в качестве цели отображения ассоциации. Мы обсудим эту тему далее в данной главе.

Все рассмотренные до сих пор стратегии отображения не требовали дополнительных соображений по поводу схемы SQL. Следующая стратегия меняет эту ситуацию.

6.3. Единая таблица для целой иерархии классов

Иерархию классов целиком можно отобразить в одну таблицу. Эта таблица будет содержать столбцы для всех полей каждого класса в иерархии. Конкретный подкласс, представляемый отдельной записью, определяется значением дополнительного столбца с селектором типа или формулой. Такой подход показан на рис. 6.2.

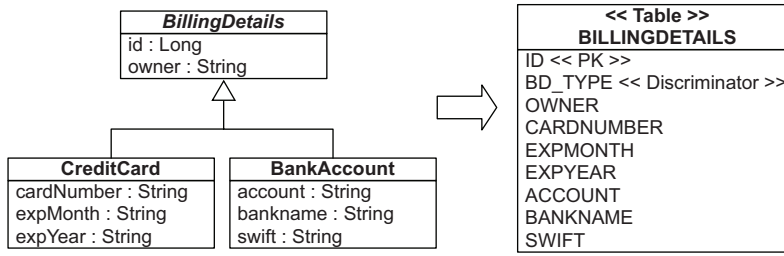


Рис. 6.2 ❖ Отображение целой иерархии классов в одну таблицу

Данная стратегия отображения – лучшая с точки зрения производительности и простоты. Это самый производительный способ представления полиморфизма – как полиморфные, так и неполиморфные запросы работают быстро, – и можно с легкостью писать запросы вручную. Можно получать отчеты на основе произвольных запросов, не применяя сложных соединений или объединений. Эволюция схемы происходит довольно просто.

Но есть одна главная проблема – целостность данных. Столбцы для свойств, объявленных в подклассах, могут содержать null. Если каждый подкласс объявляет несколько свойств, которым нельзя присваивать null, отказ от ограничения NOT NULL может стать серьезной проблемой с точки зрения корректности данных. Представьте, что приложение требует наличия даты истечения срока действия кредитной карты, но схема базы данных не способна это требование гарантировать, поскольку каждый столбец в таблице может иметь значение NULL. Простая программная ошибка в приложении может привести к некорректным данным.

Другим важным аспектом является нормализация. Вы создаете функциональную зависимость между неключевыми столбцами, нарушая третью нормальную форму. Как всегда, денормализация с целью повышения производительности может оказаться обманчивой, поскольку приходится жертвовать долговременной стабильностью, удобством сопровождения и гарантией целостности данных ради сиюминутных приобретений, которые также можно достигнуть надлежащей оп-

тимизацией планов выполнения SQL (иначе говоря, спросите своего администратора базы данных).

Для создания отображения с одной таблицей для целой иерархии классов используем стратегию наследования `SINGLE_TABLE`, как показано в листинге 6.5.

Листинг 6.5 ❖ Отображение класса `BillingDetails` с использованием стратегии `SINGLE_TABLE`

Файл: /model/src/main/java/org/jpwh/model/inheritance/singletable/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "BD_TYPE")
public abstract class BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull  ← Hibernate игнорирует эту аннотацию при генерации схемы!
    @Column(nullable = false)
    protected String owner;

    // ...
}
```

Корневой класс иерархии наследования `BillingDetails` автоматически отображается в таблицу `BILLINGDETAILS`. Наследуемые свойства суперкласса могут иметь в схеме ограничение `NOT NULL`; экземпляр каждого подкласса должен содержать какое-то значение. Особенность реализации Hibernate требует определить возможность присваивания `null`, используя аннотацию `@Column`, поскольку при генерации схемы Hibernate игнорирует аннотацию `@NotNull` из Bean Validation.

Для различения типов записей необходимо определить столбец селектора. Он не является полем сущности – Hibernate использует его для собственных нужд. Столбец называется `BD_TYPE`, а его значениями являются строки – в данном случае `"CC"` или `"BA"`. Hibernate автоматически устанавливает и извлекает значения селектора.

Если не определить столбец селектора в суперклассе, по умолчанию он будет называться `DTYPE`, а его значениями будут строки. Каждый конкретный класс в иерархии наследования может задавать свое значение селектора, как показано в определении класса `CreditCard`.

Листинг 6.6 ❖ Отображение класса `CreditCard`

Файл: /model/src/main/java/org/jpwh/model/inheritance/singletable/CreditCard.java

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
```

```
    @NotNull  ← Hibernate игнорирует эту аннотацию при генерации схемы!
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;

    // ...
}
```

Особенности Hibernate

Если не указать значения селектора явно, по умолчанию Hibernate будет использовать полное квалифицированное имя класса, если используются XML-файлы Hibernate, или простое имя сущности, если используются аннотации или XML-файлы JPA. Обратите внимание, что JPA не определяет значения по умолчанию для нестроковых селекторов; каждый производитель механизма хранения может задавать разные значения по умолчанию. Следовательно, вы всегда должны указывать значения селектора для конкретных классов.

Отметьте каждый подкласс аннотацией `@Entity`, а затем отобразите поля подкласса в столбцы таблицы `BILLINGDETAILS`. Помните, что схема не допускает ограничений `NOT NULL`, поскольку в экземпляре `BankAccount` может не оказаться свойства `expMonth` – в такой записи столбец `EXPMONTH` должен содержать значение `NULL`. Hibernate игнорирует аннотацию `@NotNull` при формировании схемы, но проверяет ее перед вставкой записи во время выполнения. Это позволяет избежать программных ошибок; никому не хочется случайно сохранить данные о кредитной карте без даты истечения срока действия (конечно, другие приложения, не соблюдающие правил, смогут поместить некорректные данные в базу).

Hibernate сформирует следующий код SQL для запроса `select bd from BillingDetails bd`:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, BD_TYPE
from
    BILLINGDETAILS
```

Для выполнения запроса к подклассу `CreditCard` Hibernate добавит ограничение на столбец селектора:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
from
    BILLINGDETAILS
where
    BD_TYPE='CC'
```

Особенности Hibernate

Иногда, особенно при работе с унаследованными схемами, просто нет возможности добавить еще один столбец селектора в таблицы сущностей. В таком случае можно применить выражение для вычисления значения селектора в каждой строке. Формулы вычисления селектора не являются частью JPA, но в Hibernate для этих целей имеется дополнительная аннотация `@DiscriminatorFormula`.

Листинг 6.7 ❖ Отображение класса `BillingDetails` с аннотацией `@DiscriminatorFormula`

Файл: /model/src/main/java/org/jpwh/model/inheritance/singletableformula/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@org.hibernate.annotations.DiscriminatorFormula(
        "case when CARDNUMBER is not null then 'CC' else 'BA' end"
)
public abstract class BillingDetails {
    // ...
}
```

В схеме отсутствует столбец селектора, поэтому, чтобы определить, что представляет конкретная запись – кредитную карту или банковский счет, данное отображение полагается на SQL-выражение `CASE/WHEN` (большинство разработчиков никогда не использовало подобного выражения SQL; если вы незнакомы с ним, обратитесь к стандарту ANSI). Результатом вычисления выражения будет литерал – `CC` или `BA`, – который должен быть объявлен в отображении соответствующего подкласса.

Недостатки такой стратегии отображения, когда одна таблица используется для целой иерархии, могут оказаться слишком серьезными для проекта – денормализованные схемы в долгосрочной перспективе могут превратиться в тяжкое бремя. Администратору базы данных это может совсем не понравиться. Следующая стратегия отображения наследования не создает подобных проблем.

6.4. Одна таблица для каждого подкласса с использованием соединений

Четвертый вариант – представление иерархии наследования в виде ассоциации по внешнему ключу в SQL. Каждый класс/подкласс, включая абстрактные классы и даже интерфейсы, объявляющий хранимые свойства, располагает собственной таблицей.

В отличие от стратегии, где каждому конкретному классу соответствует своя таблица, которую мы рассматривали первой, здесь таблица конкретной сущности

(@Entity) содержит только столбцы для ненаследуемых свойств, объявленных в самом подклассе наряду с первичным ключом, также являющимся внешним ключом таблицы суперкласса. Это проще, чем кажется; взгляните на рис. 6.3.

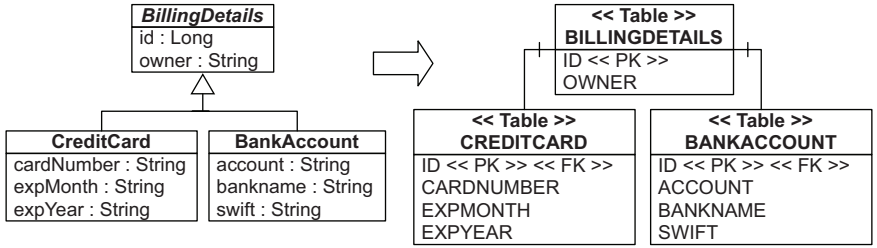


Рис. 6.3 ❖ Отображение каждого класса иерархии в собственную таблицу

При сохранении экземпляра подкласса `CreditCard` Hibernate вставляет две записи. Значения свойств, объявленных в суперклассе `BillingDetails`, сохраняются в новой записи, в таблице `BILLINGDETAILS`. Значения свойств, объявленных лишь в подклассе, сохраняются в новой записи, в таблице `CREDITCARD`. Только общий первичный ключ объединяет две эти записи. Экземпляр подкласса может быть позже извлечен из базы данных с использованием соединения таблиц подкласса и суперкласса.

Главное преимущество этой стратегии заключается в нормализации схемы SQL. Эволюция схемы и определение ограничений целостности осуществляются довольно просто. Внешний ключ, ссылающийся на таблицу конкретного подкласса, может представлять полиморфную ассоциацию с этим конкретным подклассом. Используйте стратегию наследования `JOINED`, чтобы создать отображение иерархии с отдельной таблицей для каждого подкласса.

Листинг 6.8 ❖ Отображение класса `BillingDetails` с применением стратегии отображения `JOINED`

Файл: /model/src/main/java/org/jpwh/model/inheritance/joined/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String owner;

    // ...
}
```

Корневой класс `BillingDetails` отображается в таблицу `BILLINGDETAILS`. Обратите внимание, что в этой стратегии селектор не требуется.

В подклассах не нужно указывать столбец для соединения, если столбец первичного ключа таблицы подкласса имеет (или предполагается, что имеет) то же имя, что и столбец первичного ключа в таблице суперкласса.

Листинг 6.9 ❖ Отображение класса `BankAccount` (конкретный класс)

Файл: /model/src/main/java/org/jpwh/model/inheritance/joined/BankAccount.java

```
@Entity
public class BankAccount extends BillingDetails {

    @NotNull
    protected String account;

    @NotNull
    protected String bankname;

    @NotNull
    protected String swift;

    // ...
}
```

В этой сущности отсутствует свойство-идентификатор – она автоматически наследует свойство и столбец `ID` от суперкласса, и Hibernate знает, как соединить таблицы, если потребуется извлечь экземпляры класса `BankAccount`. Конечно, можно явно задать имя столбца.

Листинг 6.10 ❖ Отображение класса `CreditCard`

Файл: /model/src/main/java/org/jpwh/model/inheritance/joined/CreditCard.java

```
@Entity
@PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
public class CreditCard extends BillingDetails {

    @NotNull
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;

    // ...
}
```

Столбцы первичных ключей в таблицах `BANKACCOUNT` и `CREDITCARD` также обладают ограничением по внешнему ключу, ссылающемуся на первичный ключ таблицы `BILLINGDETAILS`.

Hibernate использует внешнее соединение SQL для запроса `select bd from BillingDetails bd`:

```
select
    BD.ID, BD.OWNER,
    CC.EXPMONTH, CC.EXPYEAR, CC.CARDNUMBER,
    BA.ACCOUNT, BA.BANKNAME, BA.SWIFT,
    case
        when CC.CREDITCARD_ID is not null then 1
        when BA.ID is not null then 2
        when BD.ID is not null then 0
    end
from
    BILLINGDETAILS BD
    left outer join CREDITCARD CC on BD.ID=CC.CREDITCARD_ID
    left outer join BANKACCOUNT BA on BD.ID=BA.ID
```

Предложение `CASE ... WHEN` определяет наличие (либо отсутствие) строк в таблицах подклассов `CREDITCARD` и `BANKACCOUNT`, благодаря чему Hibernate может определить конкретный подкласс для каждой записи в таблице `BILLINGDETAILS`.

Для более простого запроса к подклассу, такого как `select cc from CreditCard cc`, Hibernate использует внутреннее соединение:

```
select
    CREDITCARD_ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
from
    CREDITCARD
    inner join BILLINGDETAILS on CREDITCARD_ID=ID
```

Как видите, подобную стратегию отображения уже труднее реализовать вручную – даже создавать отчеты на основе произвольных запросов становится сложнее. Важно принять этот нюанс во внимание, если вы планируете смешивать код Hibernate и код SQL, написанный вручную.

Более того, хотя эта стратегия кажется простой, наш опыт говорит, что для сложных иерархий классов производительность может оказаться неприемлемой. Запросы требуют соединения нескольких таблиц или многих последовательных операций чтения.

Наследование с соединениями и селекторами

Для реализации стратегии `InheritanceType.JOINED` Hibernate не требуется наличие в базе данных специального столбца селектора. Спецификация JPA также не содержит никаких требований на этот счет. Предложение `CASE... WHEN` в SQL-выражении `SELECT` – всего лишь ловкий способ определить тип сущности для каждой извлекаемой записи. Тем не менее некоторые примеры JPA, которые могут вам встретиться, используют и стратегию `InheritanceType.JOINED`, и отображение `@DiscriminatorColumn`. Очевидно, другие реализации JPA не используют предложения `CASE ... WHEN` и полагаются на значение селектора даже при использовании

стратегии `InheritanceType.JOINED`. Hibernate не требует наличия селектора, но может использовать объявление `@DiscriminatorColumn` даже со стратегией отображения `JOINED`. Если вы предпочитаете игнорировать отображение селектора для стратегии `JOINED` (оно игнорировалось в ранних версиях Hibernate), можно активировать параметр конфигурации `hibernate.discriminator.ignore_explicit_for_joined`.

Прежде чем показать, когда и какую стратегию выбирать, давайте рассмотрим смешение стратегий отображения наследования в одной иерархии классов.

6.5. Смешение стратегий отображения наследования

Целую иерархию классов можно отобразить с помощью стратегий `TABLE_PER_CLASS`, `SINGLE_TABLE` или `JOINED`. Но их нельзя смешивать – например, переключаться со стратегии отображения одной таблицы для целой иерархии классов с селектором на нормализованную стратегию с одной таблицей для каждого подкласса. Выбрав стратегию наследования, вы должны твердо ее придерживаться.

Однако иногда, используя некоторые приемы, все же можно переключить стратегию отображения конкретного подкласса. Например, можно отобразить иерархию классов в единственную таблицу, но для конкретного подкласса выбрать стратегию с отдельной таблицей и внешним ключом – так же, как в стратегии с таблицей для каждого подкласса. Взгляните на схему на рис. 6.4.

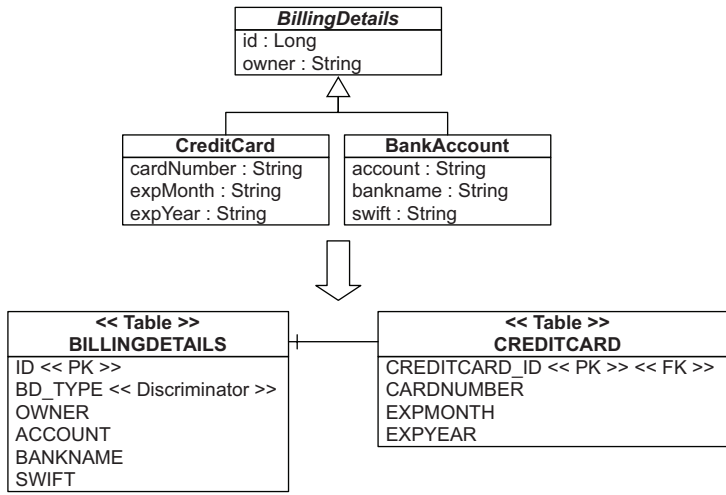


Рис. 6.4 ❖ Выделение подкласса в собственную дополнительную таблицу

Отобразим суперкласс `BillingDetails` со стратегией `InheritanceType.SINGLE_TABLE`, как мы делали это раньше. Теперь отобразим в отдельную таблицу подкласс, который нужно выделить из общей таблицы.

Листинг 6.11 ❖ Отображение класса `CreditCard`

Файл: /model/src/main/java/org/jpwh/model/inheritance/mixed/CreditCard.java

```
@Entity
@DiscriminatorValue("CC")
@SecondaryTable(
        name = "CREDITCARD",
        pkJoinColumns = @PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
)
public class CreditCard extends BillingDetails {

    @NotNull  ← JPA игнорирует эту аннотацию при генерации DDL; стратегия – SINGLE_TABLE!
    @Column(table = "CREDITCARD", nullable = false)  ← Переопределение основной таблицы
    protected String cardNumber;

    @Column(table = "CREDITCARD", nullable = false)
    protected String expMonth;

    @Column(table = "CREDITCARD", nullable = false)
    protected String expYear;

    // ...
}
```

Аннотации `@SecondaryTable` и `@Column` объединяют некоторые свойства и сообщают Hibernate, что их нужно брать из дополнительной таблицы. Все свойства, которые вы переместили в дополнительную таблицу, должны отображаться с именем этой таблицы. Для этого применяется параметр `table` аннотации `@Column`, не показанный ранее. У такого отображения есть множество применений, и вы встретитесь с ним позже в этой книге. В данном примере оно выделяет поля класса `CreditCard` из общей таблицы в таблицу `CREDITCARD`.

Столбец `CREDITCARD_ID` в то же время является первичным ключом, а также имеет ограничение по внешнему ключу, ссылающемуся на столбец `ID` единой таблицы иерархии. Если не указать название столбца первичного ключа в дополнительной таблице для соединения, по умолчанию будет выбрано имя первичного ключа единой таблицы наследования – в данном случае это `ID`.

Помните, что при выборе стратегии `InheritanceType.SINGLE_TABLE` все столбцы подклассов могут содержать значение null. Одним из преимуществ данного способа отображения является возможность гарантии целостности данных с помощью объявления ограничения `NOT NULL` для столбцов таблицы `CREDITCARD`.

Во время работы приложения Hibernate выполнит внешнее соединение для полиморфного извлечения экземпляров `BillingDetails` и всех его подклассов:

```
select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT,
    EXPMONTH, EXPYEAR, CARDNUMBER,
```

```
    BD_TYPE
from
    BILLINGDETAILS
    left outer join CREDITCARD on ID=CREDITCARD_ID
```

Вы также можете применить этот прием к остальным подклассам иерархии. Но если иерархия классов слишком обширна, внешнее соединение может стать проблемой. Некоторые СУБД (например, Oracle) ограничивают количество таблиц, участвующих в операции соединения. Для такой обширной иерархии, возможно, предпочтительнее выбрать иную стратегию извлечения, которая немедленно выполнит второй запрос SQL вместо внешнего соединения.

Особенности Hibernate

На момент написания книги переключение стратегии извлечения для данного отображения было невозможно ни при помощи аннотаций JPA, ни при помощи аннотаций Hibernate, поэтому придется отобразить класс в оригинальном XML-файле отображения Hibernate:

Файл: /model/src/main/resources/inheritance/mixed/FetchSelect.hbm.xml

```
<subclass name="CreditCard"
          discriminator-value="CC">
    <join table="CREDITCARD" fetch="select">
        ...
    </join>
</subclass>
```

До сих пор мы говорили лишь о наследовании сущностей. Несмотря на то что спецификация JPA ничего не говорит о наследовании и полиморфизме встраиваемых (`@Embeddable`) классов, Hibernate предлагает свою стратегию отображения компонентов.

Особенности Hibernate

6.6. Наследование и встраиваемые классы

Встраиваемый класс – это компонент сущности-владельца; следовательно, к ним не применимы обычные правила отображения наследования сущностей, представленные в этой главе. Используя расширенные возможности Hibrnate, можно отобразить встраиваемый класс, наследующий некоторые хранимые поля от суперкласса (или интерфейса). Рассмотрим два новых атрибута аукционного товара – вес и габариты.

Габариты товара – это ширина, высота и глубина, выраженные в некоторой системе мер и обозначающего ее символа – например, дюймов (") или сантиметров (cm). Вес товара также включает единицу измерения – например, фунты (lbs) или

килограммы (kg). Чтобы выделить общие атрибуты (название и символ) единицы измерения, можно определить суперкласс `Measurement`, общий для `Dimension` и `Weight`.

Листинг 6.12 ❖ Отображение встраиваемого абстрактного суперкласса `Measurement`

Файл: /model/src/main/java/org/jpwh/model/inheritance/embeddable/Measurement.java

```
@MappedSuperclass
public abstract class Measurement {

    @NotNull
    protected String name;

    @NotNull
    protected String symbol;

    // ...
}
```

Примените аннотацию `@MappedSuperclass` к суперклассу встраиваемого класса, который требуется отобразить, как к обычной сущности, и подклассы унаследуют свойства этого класса как хранимые свойства.

Классы `Dimensions` и `Weight` определяются как встраиваемые (`@Embeddable`). В классе `Dimensions` нужно переопределить все атрибуты суперкласса, добавив префикс столбца.

Листинг 6.13 ❖ Отображение класса `Dimensions`

Файл: /model/src/main/java/org/jpwh/model/inheritance/embeddable/Dimensions.java

```
@Embeddable
@AttributeOverrides({
        @AttributeOverride(name = "name",
                column = @Column(name = "DIMENSIONS_NAME")),
        @AttributeOverride(name = "symbol",
                column = @Column(name = "DIMENSIONS_SYMBOL"))
})
public class Dimensions extends Measurement {

    @NotNull
    protected BigDecimal depth;

    @NotNull
    protected BigDecimal height;

    @NotNull
    protected BigDecimal width;

    // ...
}
```

Без этого переопределения класс `Item`, содержащий `Dimension` и `Weight`, отображался бы в таблицу с конфликтующими названиями столбцов. Далее показан класс `Weight`; его отображение также добавляет префикс к именам столбцов (для единообразия мы решили избежать конфликта с предыдущим переопределением).

Листинг 6.14 ❖ Отображение класса `Weight`

Файл: /model/src/main/java/org/jpwh/model/inheritance/embeddable/Weight.java

```
@Embeddable
@AttributeOverrides({
        @AttributeOverride(name = "name",
                column = @Column(name = "WEIGHT_NAME")),
        @AttributeOverride(name = "symbol",
                column = @Column(name = "WEIGHT_SYMBOL"))
})
public class Weight extends Measurement {

    @NotNull
    @Column(name = "WEIGHT")
    protected BigDecimal value;

    // ...
}
```

Сущность-владелец `Item` определяет два хранимых встроенных поля.

Листинг 6.15 ❖ Отображение класса `Item`

Файл: /model/src/main/java/org/jpwh/model/inheritance/embeddable/Item.java

```
@Entity
public class Item {

    protected Dimensions dimensions;

    protected Weight weight;

    // ...
}
```

Это отображение показано на рис. 6.5. Другой способ разрешения конфликта имен заключается в переопределении названий столбцов в `Measurement` для встроенных свойств в классе `Item`, как показано в разделе 5.2. Но вместо этого мы предпочитаем переопределить их один раз во встроенных (`@Embeddable`) классах, чтобы пользователям этих классов не пришлось разрешать этого конфликта.

Подвох, которого нужно остерегаться, заключается в использовании встроенного свойства, имеющего тип абстрактного суперкласса (такого как `Measurement`), в классе сущности (таком как `Item`). Это никогда не будет работать; реализация JPA не знает, как полиморфно сохранять и загружать экземпляры `Measurement`. У него нет необходимой информации, чтобы решить, принадлежат ли экземпляры в базе данных классу `Dimension` или `Weight`, поскольку селектор отсутствует. Это значит,

что хотя встраиваемый (`@Embeddable`) класс *может* наследовать хранимые поля от класса с аннотацией `@MappedSuperclass`, ссылка *на* экземпляр не может быть полиморфной – в качестве ее типа должен выступать конкретный класс.

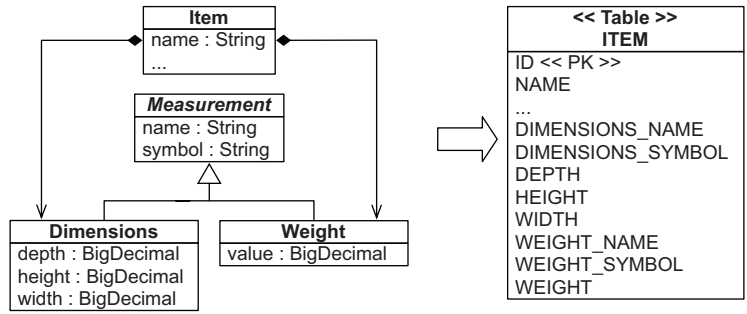


Рис. 6.5 ❖ Отображение конкретных встраиваемых классов с унаследованными полями

Сравните такой подход с альтернативной стратегией наследования для встраиваемых классов, показанной в разделе «Преобразование значений полей компонентов» в главе 5, которая поддерживает полиморфизм, но требует писать дополнительный код различения типов.

Далее мы дадим вам советы по выбору подходящей комбинации стратегий отображения для иерархий классов вашего приложения.

6.7. Выбор стратегии

Выбор правильной стратегии отображения наследования зависит от использования суперклассов иерархии сущностей. Вы должны учесть, как часто запрашиваются экземпляры суперклассов, а также наличие ассоциаций, направленных к суперклассам. Другой важный аспект – атрибуты супертипов и подтипов: отличаются ли подклассы от супертипа наличием дополнительных атрибутов или только поведением. Вот несколько практических рекомендаций.

- Если полиморфные ассоциации или запросы не требуются, выбирайте одну таблицу для каждого конкретного класса. Другими словами: если вы редко выполняете (или никогда не выполняете) запрос `select bd from BillingDetails bd` и если у вас нет классов, ссылающихся на `BillingDetails`, отдавайте предпочтение явному отображению на основе `UNION` со стратегией `InheritanceType.TABLE_PER_CLASS`, поскольку при этом сохраняется возможность последующего добавления (оптимизированных) полиморфных запросов и ассоциаций.
- Если требуются полиморфные запросы или ассоциации (ассоциация с суперклассом, а следовательно, со всеми классами иерархии и с динамиче-

ским определением конкретного класса во время выполнения), а подклассы объявляют относительно мало новых полей (особенно если основная разница между подклассами только в поведении), выбирайте стратегию `InheritanceType.SINGLE_TABLE`. Ваша цель в том, чтобы сократить количество столбцов, которые могут содержать null, и убедить себя (и администратора базы данных), что денормализованная схема не вызовет впоследствии проблем.

- Если требуются полиморфные ассоциации или запросы, а подклассы объявляют много (обязательных к заполнению) полей (подклассы отличаются в основном данными), выбирайте стратегию `InheritanceType.JOINED`. С другой стороны, в зависимости от ширины и глубины иерархии наследования и возможной стоимости соединений по сравнению с объединениями вы можете выбрать стратегию `InheritanceType.TABLE_PER_CLASS`. Такое решение требует оценки планов выполнения запросов SQL на реальных данных.

Стратегию `InheritanceType.SINGLE_TABLE` следует выбирать только для решения простых проблем. В сложных ситуациях или когда те, кто моделируют данные, настаивают на важности ограничений `NOT NULL`, и над вами довлеет нормализация, выбирайте стратегию `InheritanceType.JOINED`. В такие моменты спросите себя, не лучше ли вовсе избавиться от наследования, заменив его делегированием. Сложного наследования, как правило, следует избегать в силу разных причин, никак не связанных с хранением данных или ORM. Hibernate выступает в роли прослойки между предметной и реляционной моделями, но это не означает, что вы можете полностью игнорировать вопросы хранения при проектировании классов.

Если вы подумываете о том, чтобы смешать несколько стратегий отображения наследования, помните, что реализация неявного полиморфизма в Hibernate способна справиться с весьма экзотическими ситуациями. Также примите во внимание, что невозможно применять аннотации наследования к интерфейсам; это не предусмотрено в JPA.

Для примера рассмотрим дополнительный интерфейс в примере приложения: `ElectronicPaymentOption`. Это бизнес-интерфейс, в котором аспект хранения полностью отсутствует, за исключением того, что он может быть реализован хранимым классом, таким как `CreditCard`. Независимо от того, как вы отобразите иерархию класса `BillingDetails`, Hibernate правильно обработает следующий запрос: `select o from ElectronicPaymentOption o`. Этот запрос будет работать для хранимых классов, реализующих данный интерфейс, даже если эти классы не являются частью иерархии `BillingDetails`. Hibernate всегда знает, какие таблицы запрашивать, какие экземпляры создавать и как возвращать полиморфные результаты.

Вы можете применять все стратегии отображения к абстрактным классам. Hibernate не станет пытаться создавать экземпляр абстрактного класса, даже если вы будете выбирать его в запросе или загружать.

Мы уже несколько раз упомянули об отношении между классами `User` и `BillingDetails` и его влиянии на выбор стратегии отображения наследования. В следующем и последнем разделе этой главы мы подробно рассмотрим эту более сложную

тему полиморфных ассоциаций. Если в вашей модели такие отношения на данный момент отсутствуют, вы можете прочитать этот материал позже, когда столкнетесь с этой особенностью в своем приложении.

6.8. Полиморфные ассоциации

Полиморфизм является определяющей особенностью объектно-ориентированных языков программирования, таких как Java. Поддержка полиморфных ассоциаций и полиморфных запросов является фундаментальной особенностью таких реализаций ORM, как Hibernate. Удивительно, как далеко нам удалось забраться, не углубляясь в обсуждение полиморфизма. К счастью, на эту тему особенно нечего сказать – полиморфизм так легко использовать в Hibernate, что объяснить его будет нетрудно.

В качестве обзора мы сначала рассмотрим ассоциацию *многие к одному* (many-to-one), направленную к классу, который может иметь много подклассов, а затем отношение *один ко многим* (one-to-many). В обоих примерах классы модели предметной области будут одинаковыми; см. рис. 6.6.

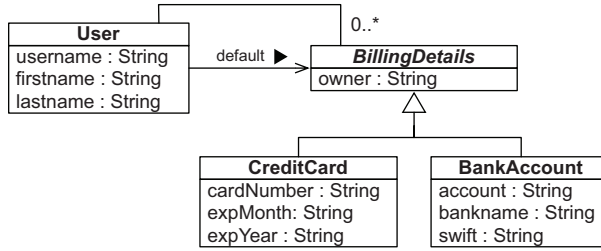


Рис. 6.6 ❖ В качестве платежных реквизитов по умолчанию у пользователя выбрана кредитная карта или банковский счет

6.8.1. Полиморфная ассоциация *многие к одному* (many-to-one)

Сначала рассмотрим поле `defaultBilling` класса `User`. Оно ссылается на единственный экземпляр `BillingDetails`, который во время выполнения может оказаться любым конкретным экземпляром этого класса.

Отобразить эту однонаправленную ассоциацию с абстрактным классом `BillingDetails` можно следующим образом:

Файл: /model/src/main/java/org/jpwh/model/inheritance/associations/manytoone/ User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @ManyToOne(fetch = FetchType.LAZY)
```

```
    protected BillingDetails defaultBilling;

    // ...
}
```

Теперь в таблице USERS появится столбец внешнего ключа для соединения DEFAULTBILLING_ID, представляющий данное отношение. В этом столбце может содержаться значение null, поскольку пользователь (User) мог пока не выбрать способ оплаты по умолчанию. Поскольку класс BillingDetails – абстрактный, ассоциация должна во время выполнения ссылаться на экземпляр одного из его подклассов – CreditCard или BankAccount. Чтобы использовать полиморфные ассоциации в Hibernate, не нужно делать ничего особенного; если целевой класс ассоциации отображается с помощью аннотаций @Entity и @Inheritance, ассоциация автоматически становится полиморфной.

Следующий код демонстрирует создание ассоциации с экземпляром подкласса CreditCard:

Файл: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
CreditCard cc = new CreditCard(
    "John Doe", "1234123412341234", "06", "2015"
);
User johndoe = new User("johndoe");
johndoe.setDefaultBilling(cc);

em.persist(cc);
em.persist(johndoe);
```

Теперь, во время навигации по ассоциации во второй единице работы, Hibernate автоматически будет извлекать экземпляры CreditCard:

Файл: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
User user = em.find(User.class, USER_ID);

user.getDefaultBilling().pay(123);  ← Вызовет метод pay() конкретного подкласса BillingDetails
```

Но имейте в виду, поскольку поле defaultBilling отображается с параметром FetchType.LAZY, Hibernate завернет цель ассоциации в прокси-объект. В этом случае вы не сможете выполнить приведение типа к конкретному классу CreditCard, и даже оператор instanceof будет вести себя странно:

Файл: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
User user = em.find(User.class, USER_ID);

BillingDetails bd = user.getDefaultBilling();

assertFalse(bd instanceof CreditCard);

// CreditCard creditCard = (CreditCard) bd;  ← Вызовет исключение ClassCastException!
```

Ссылка bd в данном случае не является экземпляром CreditCard – это специальный подкласс BillingDetails, сгенерированный во время выполнения, – прокси-класс Hibernate. Когда вызывается метод прокси-объекта, Hibernate делегирует его выполнение экземпляру CreditCard, загрузка которого отложена. Пока инициализация этого экземпляра не произошла, Hibernate не знает его подтипа – это потребовало бы обращения к базе данных, которого вы в первую очередь и пытаетесь избежать с помощью отложенной загрузки. Для безопасного преобразования типов используйте em.getReference():

Файл: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
User user = em.find(User.class, USER_ID);

BillingDetails bd = user.getDefaultBilling();

CreditCard creditCard =
    em.getReference(CreditCard.class, bd.getId());  ← Никакого SELECT

assertTrue(bd != creditCard);  ← Осторожно!
```

После вызова getReference() ссылки bd и creditCard будут ссылаться на два разных прокси-объекта, каждый из которых делегирует работу одному и тому же скрытому экземпляру CreditCard. Несмотря на это, у второго прокси-объекта будет другой интерфейс, и вы сможете вызывать такие методы, как creditCard.getExpMonth(), применимые только к этому интерфейсу (обратите внимание, что bd.getId() *вызовет* SELECT, если отобразить свойство id с использованием стратегии прямого доступа к полям).

Этих проблем можно избежать, если отказаться от отложенной загрузки, как показано в следующем примере, где применяется запрос с непосредственной загрузкой:

Файл: /examples/src/test/java/org/jpwh/test/inheritance/PolymorphicManyToOne.java

```
User user = (User) em.createQuery(
    "select u from User u " +
        "left join fetch u.defaultBilling " +
        "where u.id = :id")
    .setParameter("id", USER_ID)
    .getSingleResult();

CreditCard creditCard = (CreditCard) user.getDefaultBilling();
```

Прокси-объект не используется: экземпляр BillingDetails извлекается сразу же

Настоящий объектно-ориентированный код не должен использовать оператора instanceof или многочисленных преобразований типов. Если обнаружится, что вы вляпались в проблемы с прокси-классами, критически оцените свой проект приложения и поищите более полиморфный подход. Hibernate также поддерживает возможность внедрения в байт-код для организации отложенной загрузки через прокси; мы вернемся к стратегиям извлечения позже, в главе 12.

Таким же способом можно работать с ассоциациями *один к одному*. Но что по поводу множественных ассоциаций, таких как коллекции платежных реквизитов (`billingDetails`) для каждого пользователя (`User`)? Давайте посмотрим на них далее.

6.8.2. Полиморфные коллекции

У пользователя (`User`) могут быть ссылки на множество платежных реквизитов (`BillingDetails`), а не только заданный по умолчанию (один из нескольких будет выбираться по умолчанию, но пока мы это проигнорируем). Отобразить такую связь можно с помощью двунаправленной ассоциации *один ко многим*:

Файл: /model/src/main/java/org/jpwh/model/inheritance/associations/onetomany/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @OneToMany(mappedBy = "user")
    protected Set<BillingDetails> billingDetails = new HashSet<>();

    // ...
}
```

Далее показан владелец отношения (объявленный в предыдущем отображении с помощью параметра `mappedBy`):

Файл: /model/src/main/java/org/jpwh/model/inheritance/associations/onetomany/BillingDetails.java

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {

    @ManyToOne(fetch = FetchType.LAZY)
    protected User user;

    // ...
}
```

До сих пор в отображении этой ассоциации не встретилось ничего необычного. Иерархия классов `BillingDetails` может отображаться со стратегиями `TABLE_PER_CLASS`, `SINGLE_TABLE` или `JOINED`. Hibernate автоматически будет использовать правильные запросы SQL с оператором `JOIN` или `UNION` для загрузки коллекции элементов.

Правда, есть одно ограничение – класс `BillingDetails` не может иметь аннотацию `@MappedSuperclass`, как было показано в разделе 6.1. Он должен отображаться с аннотациями `@Entity` и `@Inheritance`.

Ассоциации с неявным полиморфизмом

Если вам потребуется отобразить ассоциацию, направленную на иерархию классов, не отображая ее явно с помощью `@Inheritance`, в качестве крайнего средства Hibernate предоставляет следующий способ. Такое отображение можно определить в XML-файле отображений Hibernate, используя элемент `<any/>`. Мы советуем найти описание этой возможности в документации Hibernate или в предыдущем издании данной книги, но старайтесь избегать ее, поскольку она может привести к появлению уродливых схем.

6.9. Резюме

- Одна таблица для каждого конкретного класса с неявным полиморфизмом – это простейшая стратегия отображения иерархий наследования сущностей, но она не очень хорошо поддерживает полиморфные ассоциации. Кроме того, разные столбцы в разных таблицах разделяют одинаковую семантику, что затрудняет эволюционирование схемы. Мы советуем применять этот подход для верхушки иерархии классов, где полиморфизм на самом деле не нужен и не предвидится изменений в суперклассе в будущем.
- Использование одной таблицы для каждого конкретного класса с операциями объединения не является обязательной стратегией, и реализации JPA могут ее не поддерживать, зато она может работать с полиморфными ассоциациями.
- Стратегия с использованием единой таблицы для всех классов иерархии является лучшей с точки зрения производительности и простоты; можно создавать отчеты на основе произвольных запросов, не используя сложных соединений и объединений, а эволюция схемы происходит довольно просто. Главной проблемой при этом является целостность данных, поскольку приходится разрешить некоторым столбцам содержать null. Другой проблемой является нормализация: данная стратегия создает функциональные зависимости между простыми столбцами, нарушая третью нормальную форму.
- Главным преимуществом стратегии одной таблицы для каждого подкласса с операциями соединения являются нормализация схемы SQL, упрощение ее эволюции и возможность определения ограничений целостности данных. Недостаток заключается в сложности ее реализации вручную и низкой производительности для сложных иерархий классов.

Глава 7

Отображение коллекций и связей между сущностями

В этой главе:

- отображение хранимых коллекций;
- коллекции основных и встраиваемых типов;
- простые ассоциации *многие к одному* и *один ко многим*.

Как следует из нашего опыта общения с сообществом пользователей Hibernate, многие разработчики, приступая к работе с Hibernate, в первую очередь пытаются отобразить *отношения родитель/потомок*. Обычно тогда они впервые и сталкиваются с коллекциями. Также они впервые начинают задумываться о различиях между сущностями и типами-значениями или просто теряются в сложности ORM.

Центральной частью ORM является управление связями между классами и отношениями между таблицами. Большинство трудностей объектно-реляционного отображения связано с коллекциями и управлением связями между сущностями. Вы можете вернуться к этой главе позже, чтобы лучше разобраться в данной теме. Мы начнем со знакомства с основными понятиями отображения коллекций и рассмотрим несколько простых примеров. После этого вы будете готовы отобразить вашу первую связь между сущностями в виде коллекции, хотя позже, в следующей главе, мы вернемся к отображению более сложных связей. Для полноты картины рекомендуем прочесть эту главу и следующую.

Главные нововведения в JPA 2

- Поддержка коллекций и словарей основных и встраиваемых типов.
- Поддержка хранимых списков, когда индекс каждого элемента хранится в дополнительном столбце в базе данных.
- Для ассоциации *один ко многим* стала доступна возможность удаления «осиротевших» объектов.

7.1. Множества, контейнеры, списки и словари с типами-значениями

Для работы с коллекциями в Java имеется богатый API, из которого можно выбирать те интерфейсы и реализации, которые более всего подходят модели предметной области. Давайте рассмотрим отображения основных коллекций, используя один и тот же пример с классами `Image` и `Item` и внося в него лишь небольшие изменения. Мы начнем с рассмотрения схемы базы данных и отображения поля коллекции в целом. Затем перейдем к выбору конкретного интерфейса коллекции и отобразим несколько реализаций: множество, контейнер (bag) идентификаторов, список, словарь и, наконец, отсортированные и упорядоченные коллекции.

7.1.1. Схема базы данных

Давайте расширим приложение CaveatEmptor, предоставив возможность добавления изображений аукционных товаров. Забудем пока об Java-коде и, сделав шаг назад, рассмотрим лишь схему базы данных.

В базе данных вам понадобится таблица `IMAGE` для хранения изображений или, быть может, только имен файлов. В таблице также должен быть столбец внешнего ключа, например `ITEM_ID`, ссылающийся на таблицу `ITEM`. Взгляните на схему, показанную на рис. 7.1.

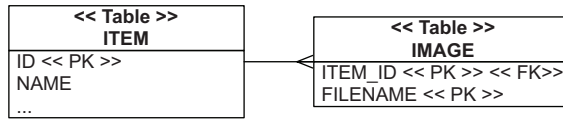


Рис. 7.1 ❖ Таблица `IMAGE` хранит имена файлов, каждый из которых ссылается на `ITEM_ID`

Это все, что касается схемы, – никаких коллекций или жизненного цикла композиции. (Вы, конечно, можете указать ограничение `ON DELETE CASCADE` для столбца внешнего ключа `ITEM_ID`. При удалении приложением записи из таблицы `ITEM` база данных автоматически удалит записи из таблицы `IMAGE`, ссылающиеся на удаляемую запись в `ITEM`. Давайте пока предположим, что это не так.)

7.1.2. Создание и отображение поля коллекции

Как бы вы отобразили таблицу `IMAGE`, используя то, что вам уже известно? Возможно, вы отобразили бы ее как класс сущности (`@Entity`) с именем `Image`. Позже в этой главе вы отобразите столбец внешнего ключа, используя поле с аннотацией `@ManyToOne`. Вам также понадобится отображение составного первичного ключа для класса сущности, как показано в разделе 9.2.2.

Нет никаких отображаемых коллекций – они попросту не нужны. Когда потребуется извлечь изображения товара, вы напишете и выполните инструкцию на

языке запросов JPA: `select img from Image img where img.item = :itemParameter`. Нет никакой необходимости хранить коллекции как таковые. Но зачем тогда может понадобиться отображать коллекции?

Коллекция, которую вы можете создать, – это `Item#images`, ссылающаяся на все изображения конкретного товара. Это поле создается и отображается для решения следующих задач:

- автоматически выполнять SQL-запрос `SELECT * from IMAGE where ITEM_ID = ?` при каждом вызове метода `someItem.getImages()`. До тех пор, пока экземпляры вашей предметной модели находятся в *управляемом* состоянии, вы можете читать данные из базы по требованию во время навигации по связям между классами. Для загрузки данных не нужно вручную писать и выполнять запросы, используя объект `EntityManager`. С другой стороны, запрос, выполняемый при обходе коллекции, всегда будет выбирать для данного товара все изображения, а не только те, что удовлетворяют условию XYZ;
- избежать сохранения экземпляров `Image` с помощью `entityManager.persist()`. Если вы отобразили коллекцию, добавление в нее объекта `Image` с помощью `someItem.getImages().add()` автоматически сделает его хранимым. Такая каскадность удобна тем, что можно сохранять экземпляры, не вызывая `EntityManager`;
- гарантировать зависимый жизненный цикл экземпляров `Image`. Во время удаления объекта `Item` Hibernate также удалит все прикрепленные объекты `Image`, выполняя дополнительное SQL-выражение `DELETE`. Вам не нужно беспокоиться о жизненном цикле изображений и удалении «осиротевших» объектов (мы предполагаем, что для внешнего ключа в базе данных отсутствует ограничение `ON DELETE CASCADE`). Реализация JPA берет управление жизненным циклом композиции на себя.

Важно осознать, что, хотя все эти преимущества и выглядят привлекательно, вы платите за них дополнительной сложностью отображения. Мы видели, как многие новички в JPA мучатся с отображениями коллекций и часто на вопрос: «Зачем ты это делаешь?», – отвечают: «Я думал, что обязательно должна использоваться коллекция».

Анализ сценария с изображениями товаров показывает, что вы только выигрываете от отображения коллекции. Изображения имеют зависимый жизненный цикл: при удалении товара все прикрепленные изображения тоже удаляются. При сохранении товара все прикрепленные изображения тоже сохраняются. При отображении товара вы обычно показываете и все его изображения, поэтому `someItem.getImages()` – это удобный метод пользовательского интерфейса. Вам не нужно повторно обращаться к уровню хранения для загрузки изображений – они уже *здесь*.

Далее мы выберем такой интерфейс и реализацию коллекции, которые лучше всего подойдут для нашей предметной модели. Давайте рассмотрим отображения основных коллекций, используя один и тот же пример с классами `Image` и `Item` и внося в него лишь небольшие изменения.

7.1.3. Выбор интерфейса коллекции

Идиома для работы с коллекциями в модели предметной области Java выгладит так:

```
<<Interface>> images = new <<Implementation>>();

// Методы доступа к свойствам
// ...
```

Для объявления типа поля используйте интерфейс, а не реализацию. Выберите подходящую реализацию и сразу же инициализируйте коллекцию; поступая так, вы избегаете появления неинициализированных коллекций. Мы не советуем инициализировать коллекции позже, в конструкторах или методах записи.

Вот типичный пример объявления параметризованного множества (`Set`):

```
Set<String> images = new HashSet<String>();
```

Обычные коллекции без параметризации

Если вы не указываете типа коллекции или ключей/значений словаря с помощью параметризации, вы должны сообщить Hibernate об этом типе (типах). К примеру, вместо `Set<String>` можно отобразить обычное множество `Set`, добавив аннотацию `@ElementCollection(targetClass=String.class)`. Вышесказанное также применимо к параметрам типов словаря (`Map`). Тип ключа словаря можно задать с помощью аннотации `@MapKeyClass`. Все примеры в этой книге используют параметризованные коллекции и словари, и вы должны поступать так же.

По умолчанию Hibernate поддерживает наиболее важные интерфейсы коллекций JDK с сохранением типичной семантики хранимых словарей и массивов. Для каждого интерфейса JDK в Hibernate имеется соответствующая реализация, и важно, чтобы вы выбирали правильную комбинацию. Hibernate обертывает коллекции, которые вы уже инициализировали в объявлении члена класса, а иногда заменяет их, если они были выбраны неверно. Это делается в том числе и для поддержки отложенной загрузки и проверки состояния элементов коллекций.

В Hibernate без расширений на выбор имеются следующие коллекции:

- свойство типа `java.util.Set`, инициализированное экземпляром `java.util.HashSet`. Порядок элементов не сохраняется, хранение повторяющихся элементов запрещено. Этот тип данных поддерживают все реализации JPA;
- свойство типа `java.util.SortedSet`, инициализированное экземпляром `java.util.TreeSet`. Данная коллекция поддерживает постоянный порядок элементов: сортировка осуществляется в памяти после загрузки данных. Упорядоченность коллекции поддерживается только в Hibernate – остальные реализации JPA могут игнорировать ее;
- свойство типа `java.util.List`, инициализированное экземпляром `java.util.ArrayList`. Hibernate сохраняет порядок следования элементов, используя

дополнительный столбец с индексами. Этот тип данных поддерживают все реализации JPA;

- свойство типа `java.util.Collection`, инициализированное экземпляром `java.util.ArrayList`. Обладает семантикой *контейнера*: может хранить повторяющиеся элементы, но их порядок не сохраняется. Этот тип данных поддерживают все реализации JPA;
- свойство типа `java.util.Map`, инициализированное экземпляром `java.util.HashMap`. Пары ключей и значений словаря могут сохраняться в базе данных. Этот тип данных поддерживают все реализации JPA;
- свойство типа `java.util.SortedMap`, инициализированное экземпляром `java.util.TreeMap`. Поддерживает порядок элементов: сортировка происходит в памяти, когда Hibernate загружает данные. Упорядоченность поддерживается только в Hibernate – остальные реализации JPA могут игнорировать ее;
- Hibernate, в отличие от JPA, поддерживает хранимые массивы. Они редко используются, и мы не станем приводить их в книге; Hibernate не может обертывать свойства-массивы, из-за чего многие преимущества коллекций, такие как отложенная загрузка, не будут работать. Используйте хранимые массивы, только если вы уверены, что отложенная загрузка вам не понадобится (есть возможность организовать отложенную загрузку массивов, но для этого потребуется перехватывать вызовы, как будет показано в разделе 12.1.3).

Особенности Hibernate

Чтобы отобразить интерфейсы и реализации коллекций, не поддерживаемые Hibernate, необходимо сообщить Hibernate о семантике ваших собственных коллекций. Точкой расширения в Hibernate является интерфейс `PersistentCollection` из пакета `org.hibernate.collection.spi`, в котором обычно выбирают для расширения один из существующих классов: `PersistentSet`, `PersistentBag` или `PersistentList`. Создать собственную коллекцию непросто, поэтому мы не советуем этого делать, если только вы не являетесь опытным пользователем Hibernate. Соответствующие примеры можно найти в исходном коде тестов для Hibernate.

Для примера с изображениями товаров мы предположим, что само изображение находится где-то в файловой системе, а в базе данных хранится лишь имя файла.

Транзакционные файловые системы

Если в базе данных SQL хранятся лишь имена файлов изображений, вы должны где-то хранить двоичные данные каждого изображения. Можно хранить данные изображений в базе SQL, в столбце типа `BLOB` (см. раздел «Двоичные типы и типы для представления больших значений» в главе 5). Если вы все же решите хранить изображения не в базе данных, а в обычных файлах, учтите, что прикладные программные интерфейсы Java для работы с файловой системой – `java.io.File` и `java.nio.file.Files` – не поддерживают транзакций. Операции файловой системы не являются частью системы транзакций (JTA): транзакция может завершиться успешно, и Hibernate запишет имя файла в базу данных SQL, но сохранение или удаление

файла может не выполниться. Эти операции нельзя откатить как атомарную единицу, а также не обеспечивается необходимой изоляции операций.
К счастью, сейчас доступны реализации транзакционных файловых систем с открытым исходным кодом для Java, такие как XADisk (см. https://xadisk.java.net). Вы можете запросто интегрировать XADisk с системным диспетчером транзакций, таким как Bitronix, который используется в примерах из этой книги. Файловые операции тогда будут включаться в транзакцию, фиксироваться и откатываться вместе с SQL-операциями Hibernate в рамках выполнения одного экземпляра `UserTransaction`.

Давайте отобразим коллекцию с именами файлов изображений для товара (`Item`).

7.1.4. Отображение множества

Простейшей реализацией является множество (`Set`) строк (`String`) с именами файлов. Добавьте свойство-коллекцию в класс `Item`, как показано в листинге 7.1.

Листинг 7.1 ❖ Изображения отображаются в виде множества строк

Файл: /model/src/main/java/org/jpwh/model/collections/setofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(
            name = "IMAGE",       ← Имя по умолчанию ITEM_IMAGES
            joinColumns = @JoinColumn(name = "ITEM_ID"))   ← Значение по умолчанию
    @Column(name = "FILENAME")    ← Имя по умолчанию IMAGES
    protected Set<String> images = new
            HashSet<String>();    ← Инициализация поля класса
    // ...
}
```

Коллекция экземпляров типов-значений должна быть отмечена JPA-аннотацией `@ElementCollection`. Без аннотаций `@CollectionTable` и `@Column` фреймворк Hibernate будет использовать имена элементов схемы. Посмотрите на схему на рис. 7.2: столбцы первичных ключей подчеркнуты.

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	FILENAME
1	foo.jpg
1	bar.jpg
1	baz.jpg
2	b.jpg

Рис. 7.2 ❖ Структура таблиц и данные для примера со множеством строк

Таблица IMAGE имеет составной первичный ключ, включающий два столбца: ITEM_ID и FILENAME. Это значит, что записи не могут повторяться: каждое изображение может быть прикреплено к каждому товару только один раз. Порядок изображений не сохраняется. Это согласуется с моделью предметной области и с коллекцией Set.

Кажется маловероятным, чтобы пользователю понадобилось прикрепить к одному товару одно и то же изображение несколько раз, но предположим, что это действительно необходимо. Какой тип отображения подойдет в этом случае?

Особенности Hibernate

7.1.5. Отображение контейнера идентификаторов

Контейнер (*bag*) – неупорядоченная коллекция, позволяющая хранить повторяющиеся элементы, подобно коллекции с интерфейсом java.util.Collection. Любопытно, но в библиотеке Java Collections реализация контейнера отсутствует. Вы инициализируете свойство объектом ArrayList, а Hibernate игнорирует индексы элементов при сохранении и загрузке элементов.

Листинг 7.2 ❖ Контейнер строк, позволяющий хранить дублирующие элементы

Файл: /model/src/main/java/org/jpwh/model/collections/bagofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @org.hibernate.annotations.CollectionId(
            columns = @Column(name = "IMAGE_ID"),
            type = @org.hibernate.annotations.Type(type = "long"),
            generator = Constants.ID_GENERATOR)
    protected Collection<String> images = new ArrayList<String>();

    // ...
}
```

Суррогатный первичный ключ допускает повторение элементов

❶ Столбец суррогатного первичного ключа

❸ Настройка первичного ключа

❷ Аннотация, доступная только в Hibernate

В JDK отсутствует реализация BagImpl

Это отображение выглядит сложнее: вы не можете использовать прежнюю схему. Для таблицы коллекции IMAGE требуется определить другой первичный ключ, допускающий повторение значений FILENAME для каждого ITEM_ID. С этой целью вводится столбец суррогатного первичного ключа с именем IMAGE_ID ❶ и применяется аннотация, доступная только в Hibernate ❷, которая определяет способ генерации первичного ключа ❸. Если вы забыли, как работают генераторы ключей, прочтите раздел 4.2.4.

Модифицированная схема показана на рис. 7.3.

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

IMAGE_ID	ITEM_ID	FILENAME
1	1	foo.jpg
2	1	bar.jpg
3	1	baz.jpg
4	1	baz.jpg
5	2	b.jpg

Рис. 7.3 ❖ Столбец суррогатного первичного ключа для контейнера со строками

Возникает интересный вопрос: сможете ли вы сказать, как эти таблицы отображаются в Java, глядя только на эту схему? Таблицы `ITEM` и `IMAGE` выглядят одинаково: каждая имеет столбец суррогатного первичного ключа и прочие нормализованные столбцы. Каждая таблица могла бы отображаться с помощью класса сущности (`@Entity`). Мы решили использовать функциональность, доступную в JPA, отобразив коллекцию в таблицу `IMAGE`, несмотря на жизненный цикл, характерный для композиции. В сущности, это решение о том, что все, что нужно для этой таблицы, – это некоторые предопределенные запросы и правила работы с данными, а не более обобщенное отображение `@Entity`. Принимая подобное решение, вы должны быть уверены в его причинах и последствиях.

Следующий способ отображения сохраняет порядок изображений в списке.

7.1.6. Отображение списка

Если прежде вам не приходилось пользоваться механизмами ORM, хранимый список может показаться мощной идеей; представьте, сколько усилий потребовалось бы для сохранения и загрузки объекта `java.util.List<String>`, используя JDBC и SQL. При вставке элемента в середину списка, в зависимости от реализации, список сдвигает все последующие элементы вправо и переустанавливает указатели. Если удалить элемент из середины списка, произойдет что-то еще и т. д. Если механизм ORM может делать все это для записей в базе данных автоматически, это делает хранимый список более привлекательным, чем он есть на самом деле.

Как отмечалось в разделе 3.2.4, в первый момент возникает желание сохранить элементы в порядке их ввода пользователем. Нередко эти данные приходится показывать позже в таком же порядке. Но если будет применено другое условие сортировки, например по времени создания записи, вам придется сортировать данные в запросе, не сохраняя их порядка. А что, если порядок вывода изменится? Порядок вывода практически никогда не является существенной характеристикой данных. Поэтому дважды подумайте, перед тем как отображать хранимый список `List`; Hibernate не настолько умен, как может показаться, что наглядно демонстрируют следующие примеры.

Для начала модифицируем сущность `Item` и ее свойство-коллекцию.

Листинг 7.3 ❖ Хранимый список, поддерживающий порядок следования элементов в базе данных

Файл: /model/src/main/java/org/jpwh/model/collections/listofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @OrderColumn  ← Позволяет сохранять порядок; имя по умолчанию: IMAGES_ORDER
    @Column(name = "FILENAME")
    protected List<String> images = new ArrayList<String>();

    // ...
}
```

В этом примере появилась новая аннотация: `@OrderColumn`. В данном столбце хранятся индексы элементов хранимого списка, начиная с нуля. Обратите внимание, что Hibernate сохраняет индекс в базе данных непрерывным и ожидает, что он будет оставаться таким же. Если возникнут пропуски, при загрузке и создании объекта `List` Hibernate добавит соответствующие элементы со значением `null`. Посмотрите на схему на рис. 7.4.

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	IMAGES_ORDER	FILENAME
1	0	foo.jpg
1	1	bar.jpg
1	2	baz.jpg
1	3	baz.jpg
2	0	b1jpg
2	1	b2.jpg

Рис. 7.4 ❖ Таблица коллекции хранит позицию каждого элемента в списке

Первичный ключ таблицы `IMAGE` состоит из `ITEM_ID` и `IMAGES_ORDER`. Это позволяет хранить одинаковые значения `FILENAME`, что согласуется с семантикой типа `List`.

Ранее мы сказали, что Hibernate не так умен, как может показаться. Посмотрим, как выполнится изменение списка из трех элементов A, B и C, расположенных именно в таком порядке. Что произойдет при удалении A из списка? Hibernate выполнит одну SQL-инструкцию `DELETE` для этой строки. Затем две инструкции `UPDATE` для B и C, сдвигая их позиции влево, чтобы заполнить пропущенный индекс. Для каждого элемента справа от удаляемого Hibernate выполнит выражение `UPDATE`. Если написать SQL-запрос вручную, можно обойтись одной инструкцией `UPDATE`. То же верно и для вставки в середину списка. Hibernate сдвинет все элементы справа на одну позицию. Hibernate хотя бы сообразит выполнить одно выражение `DELETE` при вызове метода списка `clear()`.

Теперь предположим, что помимо имени файла изображение товара имеет название, которое вводится пользователем. Смоделировать это в Java можно, например, применив словарь (map).

7.1.7. Отображение словаря

И снова внесем небольшое изменение в Java-класс, чтобы использовать свойство типа `Map`.

Листинг 7.4 ❖ Хранимый словарь с ключами и значениями

Файл: /model/src/main/java/org/jpwh/model/collections/mapofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")   <-- ❶ Отображает ключ
    @Column(name = "IMAGENAME")        <-- ❷ Отображает значение
    protected Map<String, String> images = new HashMap<String, String>();

    // ...
}
```

Каждый элемент словаря – это пара ключ/значение. Здесь ключ словаря отображается с помощью аннотации `@MapKeyColumn` в столбец `FILENAME` ❶, а значение – в столбец `IMAGENAME` ❷. Это значит, что пользователь может добавить файл только раз, потому что словарь (`Map`) не допускает повторения ключей.

Как видно из схемы на рис. 7.5, столбец первичного ключа таблицы коллекции состоит из столбцов `ITEM_ID` и `FILENAME`. В качестве типа ключа словаря в примере выбран `String`, но Hibernate поддерживает все основные типы, такие как `BigDecimal` и `Integer`. Если ключ является Java-перечислением (`enum`), вы должны использовать `@MapKeyEnumerated`. С любым типом для представления времени, таким как `java.util.Date`, используйте аннотацию `@MapKeyTemporal`. Мы уже обсуждали эти возможности в разделах 5.1.6 и 5.1.7, хотя и не в контексте отображения коллекций.

Словарь из предыдущего примера был неупорядоченным. Что необходимо сделать, чтобы сохранить словарь, отсортированный по имени файла?

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	FILENAME	IMAGENAME
1	foo.jpg	Foo
1	bar.jpg	Bar
1	baz.jpg	Baz
2	b1.jpg	B1
2	b2.jpg	B2

Рис. 7.5 ❖ Таблица для отображения словаря, в котором ключи и значения являются строками

Особенности Hibernate

7.1.8. Отсортированные и упорядоченные коллекции

В английском языке слова *sorted* (отсортированный) и *ordered* (упорядоченный) имеют разное значение, когда речь заходит о хранимых коллекциях Hibernate. Коллекции *сортируются* в памяти, с использованием Java-компараторов (функций сравнения). *Упорядочивание* коллекции производится при загрузке из базы данных, с использованием предложения `ORDER BY` в запросе SQL.

Давайте сделаем словарь с изображениями отсортированным. Для этого нужно изменить Java-свойство и его отображение.

Листинг 7.5 ❖ Сортировка записей словаря в памяти с помощью компаратора

Файл: /model/src/main/java/org/jpwh/model/collections/sortedmapofstrings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.SortComparator(ReverseStringComparator.class)
    protected SortedMap<String, String> images =
        new TreeMap<String, String>();

    // ...
}
```

Отсортированные коллекции – это особенность Hibernate, поэтому здесь использована аннотация `org.hibernate.annotations.SortComparator` с одной из реализаций `java.util.Comparator`. Мы не будем показывать этот простейший класс – он просто сортирует строки в обратном порядке.

Как и в следующих примерах, схема базы данных не меняется. Если вам нужно вспомнить, как выглядит схема, посмотрите иллюстрации в предыдущем разделе.

Отобразим `java.util.SortedSet`, как показано далее.

Листинг 7.6 ❖ Сортировка элементов множества в памяти с помощью `String#compareTo()`

Файл: /model/src/main/java/org/jpwh/model/collections/sortedsetof-strings/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @org.hibernate.annotations.SortNatural
```

```
    protected SortedSet<String> images = new TreeSet<String>();

    // ...
}
```

Здесь применяется естественная сортировка, использующая метод `String#compareTo()`.

К сожалению, вы не сможете отсортировать контейнер – не существует такого типа, как `TreeBag`. Порядок элементов определяется их индексами.

С другой стороны, вместо применения интерфейсов `Sorted*` и сортировки коллекции в памяти ее элементы можно сразу извлекать из базы данных в правильном порядке. Вместо типа `java.util.SortedSet` в следующем примере выбран `java.util.LinkedHashSet`.

Листинг 7.7 ❖ `LinkedHashSet` сохраняет порядок вставки при обходе

Файл: /model/src/main/java/org/jpwh/model/collections/setofstringsorderby/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    // @javax.persistence.OrderBy
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    protected Set<String> images = new LinkedHashSet<String>();

    // ...
}
```

Поддерживает единственный способ упорядочивания «FILENAME asc»

Класс `LinkedHashSet` сохраняет постоянный порядок обхода элементов, и Hibernate заполнит его в правильном порядке при загрузке коллекции. Для этого Hibernate добавит предложение `ORDER BY` в SQL-выражение, загружающее коллекцию. Это SQL-предложение должно объявляться с помощью аннотации `@org.hibernate.annotations.OrderBy`. В ней можно даже вызывать функции SQL, такие как `@OrderBy("substring(FILENAME, 0, 3) desc")`, которая отсортирует коллекцию по первым трем буквам в именах файлов. Но будьте внимательны и убедитесь, что СУБД поддерживает вызываемую функцию SQL. Кроме того, можно применить синтаксис SQL:2003, например `ORDER BY ... NULLS FIRST|LAST`, и Hibernate автоматически преобразует его в диалект, поддерживаемый СУБД.

@OrderBy из Hibernate или @OrderBy из JPA

Аннотацию `@org.hibernate.annotations.OrderBy` можно применить к любой коллекции; ее параметр – это обычный фрагмент SQL-кода, который Hibernate добавит в SQL-выражение, загружающее коллекцию. В Java Persistence есть похожая аннотация – `@javax.persistence.OrderBy`. Она принимает единственный параметр – `someProperty`

DESC|ASC. Элементы типа String или Integer не имеют свойств. Следовательно, если применить JPA-аннотацию @OrderBy к коллекции простого типа, как в предыдущем примере с Set<String>, согласно спецификации упорядочение будет производиться по значениям объектов простого типа. Это значит, что вы не можете поменять порядок: в предыдущем примере порядок всегда соответствовал бы условию FILENAME asc. Мы будем использовать аннотации JPA позже, в разделе 7.2.2, когда в классе значения элемента появятся сохраняемые поля и его тип будет отличаться от простого/скалярного.

В следующем примере показано такое же упорядочение во время загрузки, но с отображением контейнера.

Листинг 7.8 ❖ Тип ArrayList сохраняет постоянный порядок при обходе

Файл: /model/src/main/java/org/jpwh/model/collections/bagofstringsorderby/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @org.hibernate.annotations.CollectionId(
            columns = @Column(name = "IMAGE_ID"),
            type = @org.hibernate.annotations.Type(type = "long"),
            generator = Constants.ID_GENERATOR)
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    protected Collection<String> images = new ArrayList<String>();

    // ...
}
```

Суррогатный первичный ключ допускает дублирование элементов

Наконец, упорядоченные пары ключей и значений можно загрузить с помощью LinkedHashMap.

Листинг 7.9 ❖ LinkedHashMap хранит упорядоченные пары ключей и значений

Файл: /model/src/main/java/org/jpwh/model/collections/mapofstringsorderby/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    protected Map<String, String> images = new LinkedHashMap<String, String>();

    // ...
}
```

Помните, что элементы упорядоченных коллекций располагаются в необходимом порядке только во время загрузки. После добавления или удаления элементов порядок обхода коллекции может отличаться от упорядочивания по имени файла; эти коллекции ведут себя как обычные связанные множества, словари и списки.

В действующей системе вам, скорее всего, потребуется что-то большее, чем просто название изображения и имя файла. Возможно, придется создать класс `Image` для хранения дополнительной информации. А это может послужить превосходным примером работы с коллекцией компонентов.

7.2. Коллекции компонентов

В разделе 5.2 мы уже отображали встраиваемые компоненты: свойство `address` класса `User`. Но та ситуация отличается от текущей, потому что объект `Item` может иметь много ссылок на экземпляры `Image`, как показано на рис. 7.6. Связь на диаграмме UML является композицией (закрашенный ромбик); следовательно, объект `Image` связан с жизненным циклом объекта `Item`.

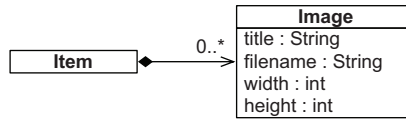


Рис. 7.6 ❖ Коллекция компонентов `Image` класса `Item`

В листинге 7.10 показан новый встраиваемый класс `Image`, описывающий все интересующие нас свойства изображения.

Листинг 7.10 ❖ Инкапсуляция всех полей изображения

Файл: /model/src/main/java/org/jpwh/model/collections/setofembeddables/Image.java

```
@Embeddable
public class Image {

    @Column(nullable = false)
    protected String title;

    @Column(nullable = false)
    protected String filename;

    protected int width;

    protected int height;

    // ...
}
```

Во-первых, обратите внимание, что все свойства являются обязательными (с ограничением `NOT NULL`). Свойства для хранения размеров не могут иметь зна-

чения null, поскольку хранят значения простых типов. Во-вторых, мы должны подумать, как база данных и Java-приложение будут сравнивать два изображения.

7.2.1. Равенство экземпляров компонентов

Предположим, что мы сохранили несколько объектов `Image` в коллекции `HashSet`. Известно, что множества не хранят повторяющихся элементов. Как же оно их определяет? Коллекция `HashSet` вызывает метод `equals()` для каждого объекта `Image`, который добавляется во множество `Set`. (Очевидно, она также вызывает метод `hashCode()`, чтобы получить хэш-код.) Сколько изображений окажется в следующей коллекции?

```
someItem.getImages().add(new Image(
        "Foo", "foo.jpg", 640, 480
));
someItem.getImages().add(new Image(
        "Bar", "bar.jpg", 800, 600
));
someItem.getImages().add(new Image(
        "Baz", "baz.jpg", 1024, 768
));
someItem.getImages().add(new Image(
        "Baz", "baz.jpg", 1024, 768
));
assertEquals(someItem.getImages().size(), 3);
```

Вы ожидали четыре изображения вместо трех? Вы правы – обычная проверка на равенство в Java использует идентичность объектов. Метод `java.lang.Object#equals()` сравнивает два экземпляра как `a==b`. Используя такое сравнение, в коллекции у вас окажутся четыре экземпляра `Image`. Ясно, что в данном примере «правильный» ответ – три.

Класс `Image` не должен полагаться на идентичность в Java, поэтому мы должны переопределить методы `equals()` и `hashCode()`.

Листинг 7.11 ❖ Собственное определение равенства с помощью методов `equals()` и `hashCode()`

Файл: /model/src/main/java/org/jpwh/model/collections/setofembeddables/Image.java

```
@Embeddable
public class Image {

    @Override
    public boolean equals(Object o) {  ← ❶ Проверка на равенство
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Image other = (Image) o;

        if (!title.equals(other.title)) return false;
```

```
        if (!filename.equals(other.filename)) return false;
        if (width != other.width) return false;
        if (height != other.height) return false;

        return true;
    }

    @Override
    public int hashCode() {   ← ❷ Должен быть симметричным
        int result = title.hashCode();
        result = 31 * result + filename.hashCode();
        result = 31 * result + width;
        result = 31 * result + height;
        return result;
    }

    // ...
}
```

Здесь метод `equals()` ❶ сравнивает все значения одного объекта `Image` со значениями другого объекта `Image`. Если совпали значения всех свойств, значит, изображения одинаковые. Метод `hashCode()` ❷ должен быть симметричным – если два экземпляра равны, они должны иметь одинаковые хэш-коды.

Почему мы не переопределяли функцию сравнения при отображении адреса (`Address`) пользователя (`User`) в разделе 5.2? По правде говоря, нам, скорее всего, стоило это сделать. Единственное оправдание – отсутствие каких-либо проблем при применении обычного сравнения Java-идентичности, пока мы не добавляем встраиваемых компонентов во множество (`Set`) и не используем их в качестве ключей словаря (`Map`). Иначе пришлось бы переопределять метод `equals()` и сравнивать значения объектов, а не их идентичности. Эти методы желательно переопределять для всех встраиваемых (`@Embeddable`) классов; все экземпляры типов-значений должны сравниваться «по значению».

Рассмотрим теперь первичный ключ в базе данных: Hibernate сгенерирует схему, объединяющую в составной первичный ключ все столбцы таблицы `IMAGE`, которые не могут содержать null. Столбцы не могут содержать null, поскольку невозможно индентифицировать то, что неизвестно. Это согласуется с процедурой определения равенства в классе Java. Схему вы увидите в следующем разделе, а также найдете больше подробностей о первичных ключах.

> **ПРИМЕЧАНИЕ** Необходимо отметить следующую особенность генератора схемы Hibernate: если отметить свойство встраиваемого класса аннотацией `@NotNull` вместо `@Column(nullable=false)`, Hibernate не сгенерирует ограничения `NOT NULL` для столбца таблицы коллекции. Проверка экземпляра механизмом Bean Validation будет работать, как полагается; только в базе данных не будет правил ограничения целостности. Используйте аннотацию `@Column(nullable=false)`, когда встраиваемый класс отображается в коллекции, а свойство должно быть частью первичного ключа.

Итак, класс компонента готов, и его можно использовать в отображениях коллекций.

7.2.2. Множество компонентов

Множество (`Set`) компонентов отображается, как показано далее.

Листинг 7.12 ❖ Множество (`Set`) встраиваемых компонентов с переопределением

Файл: /model/src/main/java/org/jpwh/model/collections/setofembeddables/Item.java

```
@Entity
public class Item {

    @ElementCollection  ← ❶ Обязательная аннотация
    @CollectionTable(name = "IMAGE")  ← ❷ Переопределяет имя таблицы коллекции
    @AttributeOverride(
            name = "filename",
            column = @Column(name = "FNAME", nullable = false)
    )
    protected Set<Image> images = new HashSet<Image>();

    // ...
}
```

Как и раньше, аннотация `@ElementCollection` ❶ обязательна. По объявлению параметризованной коллекции Hibernate автоматически определит, что целевым типом коллекции является встраиваемый (`@Embeddable`) тип.

Аннотация `@CollectionTable` ❷ переопределяет имя таблицы по умолчанию, которая иначе называлась бы `ITEM_IMAGES`.

Отображение класса `Image` определяет столбцы таблицы коллекции. Так же как в случае с единственным значением встраиваемого типа, для настройки отображения без изменения целевого встраиваемого класса можно применить аннотацию `@AttributeOverride`. Взгляните на схему базы данных на рис. 7.7.

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	TITLE	FNAME	WIDTH	HEIGHT
1	Foo	foo.jpg	640	480
1	Bar	bar.jpg	800	600
1	Baz	baz.jpg	1024	768
2	B	b.jpg	640	480

Рис. 7.7 ❖ Таблица с данными для примера с коллекцией компонентов

Поскольку отображается множество, первичный ключ таблицы коллекции будет составлен из столбца внешнего ключа `ITEM_ID` и всех «встроенных» столбцов, которые не могут содержать null: `TITLE`, `FNAME`, `WIDTH` и `HEIGHT`.

Значение ITEM_ID не используется в переопределенных методах equals() и hashCode() класса Image, как было показано в предыдущем разделе. Следовательно, смешав изображения различных товаров в одном множестве, вы столкнетесь с проблемой определения равенства на Java-уровне. Очевидно, в базе данных можно отличить изображения разных товаров, поскольку идентификатор товара участвует в проверках равенства первичных ключей.

Чтобы использовать объект Item в процедуре проверки равенства класса Image для соответствия проверке равенства первичного ключа в базе данных, необходимо добавить свойство Image#item. Это простой указатель на сущность-владельца, предоставляемый Hibernate во время загрузки экземпляров Image:

Файл: /model/src/main/java/org/jpwh/model/collections/setofembeddables/Image.java

```
@Embeddable
public class Image {

    @org.hibernate.annotations.Parent
    protected Item item;

    // ...
}
```

Теперь вы сможете использовать родительский экземпляр Item в реализации методов equals() и hashCode() и выполнять сравнение, например так: this.getItem().getId().equals(other.getItem().getId()). Но будьте осторожны: если объект Item еще не был сохранен, он не имеет идентификатора; мы глубже изучим эту проблему далее в разделе 10.3.2.

Чтобы упорядочить элементы во время загрузки и сохранить постоянный порядок обхода в LinkedHashSet, используйте JPA-аннотацию @OrderBy:

Файл: /model/src/main/java/org/jpwh/model/collections/setofembeddablesor-derby/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @OrderBy("filename, width DESC")
    protected Set<Image> images = new LinkedHashSet<Image>();

    // ...
}
```

Аргументы аннотации @OrderBy – это список полей класса Image, за которыми следует либо строка ASC для упорядочивания по возрастанию, либо DESC для упорядочивания по убыванию. По умолчанию упорядочение происходит по возрастанию, следовательно, в данном примере элементы упорядочены (по возрастанию) сначала по имени файла, а затем по ширине изображения. Обратите внимание на отличия от аннотации @org.hibernate.annotations.OrderBy, параметр которой является обычным SQL-предложением, как было показано в разделе 7.1.8.

Возможно, что объявление всех свойств класса `Image` как `@NotNull` – это не то, что вам надо. Если какое-либо из полей является необязательным, вам понадобится определить другой первичный ключ для таблицы коллекции.

7.2.3. Контейнер компонентов

Вы уже применяли аннотацию `@org.hibernate.annotations.CollectionId` ранее для добавления столбца суррогатного ключа в таблицу коллекции. Но типом коллекции был не `Set`, а `Collection`, т. е. контейнер. Это согласуется с обновленной схемой: если у вас есть столбец суррогатного первичного ключа, «значения элементов» могут повторяться. Давайте разберем это на примере.

Во-первых, класс `Image` теперь может иметь свойства со значениями null:

Файл: /model/src/main/java/org/jpwh/model/collections/bagofembeddables/Image.java

```
@Embeddable
public class Image {

    @Column(nullable = true)  ← Может содержать null, если задан суррогатный первичный ключ
    protected String title;

    @Column(nullable = false)
    protected String filename;

    protected int width;

    protected int height;

    // ...
}
```

Не забудьте принять во внимание необязательность значения `title` объекта `Image` в переопределяемых методах `equals()` и `hashCode()`, когда будете сравнивать экземпляры «по значению».

Далее показано отображение контейнера в классе `Item`:

Файл: /model/src/main/java/org/jpwh/model/collections/bagofembeddables/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @org.hibernate.annotations.CollectionId(
            columns = @Column(name = "IMAGE_ID"),
            type = @org.hibernate.annotations.Type(type = "long"),
            generator = Constants.ID_GENERATOR)
    protected Collection<Image> images = new ArrayList<Image>();

    // ...
}
```

Особенности Hibernate

Так же как в разделе 7.1.5, с помощью аннотации `@org.hibernate.annotations.CollectionId` объявлен дополнительный столбец суррогатного первичного ключа `IMAGE_ID`. Схема базы данных показана на рис. 7.8.

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

IMAGE_ID	ITEM_ID	TITLE	FILENAME	WIDTH	HEIGHT
1	1	Foo	foo.jpg	640	480
2	1		bar.jpg	800	600
3	1	Baz	baz.jpg	1024	768
4	1	Baz	baz.jpg	1024	768
5	2	B	b.jpg	640	480

Рис. 7.8 ❖ Таблица коллекции компонентов со столбцом суррогатного первичного ключа

Поле `title` экземпляра `Image` с идентификатором 2 содержит `null`.

Далее мы увидим другой способ изменения первичного ключа таблицы коллекции с помощью словаря (`Map`).

7.2.4. Словарь с компонентами в качестве значений

Если хранить экземпляры `Image` в коллекции `Map`, на роль ключа можно выбрать имя файла:

Файл: /model/src/main/java/org/jpwh/model/collections/mapofstringsembeddables/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")   ← Необязательная аннотация; имя по умолчанию IMAGES_KEY
    protected Map<String, Image> images = new HashMap<String, Image>();

    // ...
}
```

Теперь первичный ключ таблицы коллекции состоит из столбца внешнего ключа `ITEM_ID` и ключа словаря `FILENAME`, как показано на рис. 7.9.

Встраиваемый класс `Image` отображает оставшиеся столбцы, которые могут содержать null:

Файл: /model/src/main/java/org/jpwh/model/collections/mapofstringsembeddables/Image.java

```
@Embeddable
public class Image {
```

```
    @Column(nullable = true)  ← Может содержать null; не входит в состав первичного ключа
    protected String title;

    protected int width;

    protected int height;

    // ...
}
```

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	FILENAME	TITLE	WIDTH	HEIGHT
1	foo.jpg	Foo	640	480
1	bar.jpg		800	600
1	baz.jpg	Baz	1024	768
2	b.jpg	B	640	480

Рис. 7.9 ❖ Таблицы базы данных для словаря компонентов

В предыдущем примере значениями элементов словаря были экземпляры класса встраиваемого компонента, а ключами – обычные строки. Теперь попробуем использовать в качестве ключей и значений встраиваемые типы.

7.2.5. Компоненты в роли ключей словаря

В последнем примере рассмотрим отображение словаря (`Map`), ключи и значения которого хранят значения встраиваемых типов данных, как показано на рис. 7.10.

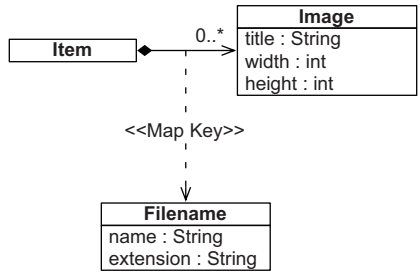


Рис. 7.10 ❖ Класс `Item` содержит коллекцию `Map` с ключами типа `Filename`

Для представления имени файла вместо строки можно выбрать пользовательский тип, как показано далее.

Листинг 7.13 ❖ Представление имени файла пользовательским типом

Файл: /model/src/main/java/org/jpwh/model/collections/mapofembeddables/Filename.java

```
@Embeddable
public class Filename {
```

```
    @Column(nullable = false)
    protected String name;

    @Column(nullable = false)
    protected String extension;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Filename filename = (Filename) o;

        if (!extension.equals(filename.extension)) return false;
        if (!name.equals(filename.name)) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = name.hashCode();
        result = 31 * result + extension.hashCode();
        return result;
    }

    // ...
}
```

Должны иметь ограничение NOT NULL: части первичного ключа

Чтобы использовать этот класс в качестве ключей словаря, отображаемые столбцы в базе данных не должны поддерживать хранения значения null, поскольку являются частями составного первичного ключа. Также необходимо переопределить методы `equals()` и `hashCode()`, так как ключи словаря являются элементами множества и каждый объект `Filename` должен быть уникальным в пределах множества ключей.

Специальных аннотаций для отображения коллекции не требуется.

Файл: /model/src/main/java/org/jpwh/model/collections/mapofembeddables/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    protected Map<Filename, Image> images = new HashMap<Filename, Image>();

    // ...
}
```

Аннотации `@MapKeyColumn` и `@AttributeOverrides` здесь не могут использоваться; когда роль ключа играет встраиваемый (`@Embeddable`) класс, эти аннотации не окажут никакого влияния. Составной первичный ключ таблицы `IMAGE` включает столбцы `IMAGE_ID`, `NAME` и `EXTENSION`, как показано на рис. 7.11.

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	NAME	EXTENSION	TITLE	WIDTH	HEIGHT
1	foo	jpg	Foo	640	480
1	bar	jpg		800	600
1	baz	jpg	Baz	1024	768
2	b	jpg	B	640	480

Рис. 7.11 ❖ Таблицы базы данных для коллекции `Map` с экземплярами `Image` и ключами типа `Filename`

Такой встраиваемый класс, как `Image`, не ограничен свойствами простых типов. Вы уже видели на примере класса `City` в классе `Address`, как можно делать компоненты вложенными. Можно извлечь и инкапсулировать свойства `width` и `height` класса `Image` в новом классе `Dimensions`.

Встраиваемый класс также может содержать коллекции.

7.2.6. Коллекции во встраиваемых компонентах

Предположим, что нужно хранить список контактов для каждого адреса (`Address`). Это простое множество типа `Set<String>` во встраиваемом классе:

Файл: /model/src/main/java/org/jpwh/model/collections/embeddablesetofstrings/Address.java

```
@Embeddable
public class Address {

    @NotNull
    @Column(nullable = false)
    protected String street;

    @NotNull
    @Column(nullable = false, length = 5)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String city;

    @ElementCollection
    @CollectionTable(
            name = "CONTACT",                                  ← По умолчанию USER_CONTACTS
            joinColumns = @JoinColumn(name = "USER_ID"))       ← Значение по умолчанию
    @Column(name = "NAME", nullable = false)                   ← По умолчанию – CONTACTS
    protected Set<String> contacts = new HashSet<String>();
    // ...
}
```

Единственная обязательная аннотация – `@ElementCollection`; таблица и столбцы могут называться по умолчанию. Взгляните на схему на рис. 7.12: столбец `USER_ID` имеет ограничение по внешнему ключу, ссылающемуся на таблицу сущ-

ности-владельца USERS. Первичный ключ таблицы коллекции состоит из столбцов USER_ID и NAME, что не позволяет дублировать элементы в соответствии с семантикой коллекции Set.

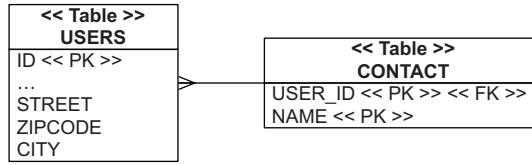


Рис. 7.12 ❖ Ограничение столбца USER_ID по внешнему ключу, ссылающемуся на таблицу USERS

Вместо коллекции Set можно выбрать отображение списка, контейнера или словаря, параметризованных простыми типами. Также Hibernate поддерживает коллекции встраиваемых типов, поэтому вместо хранения контактной информации в строках можно написать встраиваемый класс Contact и добавить в класс Address коллекцию с типом Contact.

Несмотря на то что Hibernate дает большую гибкость в выборе способов отображения компонентов и хорошо детализированных моделей, помните, что код читают гораздо чаще, чем пишут. Подумайте о следующем программисте, который будет поддерживать все это через несколько лет.

Пришло время переключить наше внимание на связи между сущностями, в частности на простые связи *многие к одному* и *один ко многим*.

7.3. Отображение связей между сущностями

В начале главы мы пообещали обсудить отношения родитель/потомок. До сих пор мы отображали только сущность Item. Пусть она будет родителем. В ней содержится коллекция потомков – экземпляров Image. Термин *родитель/потомок* подразумевает некоторую зависимость жизненных циклов, поэтому вполне подойдет коллекция строк или встраиваемых компонентов. Потомки целиком зависят от родителей: они сохраняются, обновляются и удаляются всегда только вместе с родителем, а не сами по себе. Вот мы и отобразили отношение родитель/потомок! В роли родителя выступила сущность, а множеством ее детей были экземпляры типа-значения.

Теперь отобразим отношение другого рода: связь между двумя классами сущностей. Их экземпляры существуют независимо. Один объект может сохраняться, обновляться и удаляться, никак не влияя на другой. Естественно, *иногда* возникают зависимости и между экземплярами сущностей. Нам нужен более тонкий контроль над тем, как отношение между двумя классами влияет на состояние экземпляров не полностью зависимых (встроенных) типов. Но не выходим ли мы при этом за рамки отношения родитель/потомок? Получается, что понятие *родитель/потомок* очень расплывчатое, и каждый определяет его по-своему. Мы постара-

емся больше не использовать этого термина, полагаясь на более точную или, по крайней мере, достаточно конкретную терминологию.

В следующих разделах мы будем рассматривать одно и то же отношение: между классами сущностей `Item` и `Bid`, как показано на рис. 7.13. Между классами `Bid` и `Item` определена связь *многие к одному*.

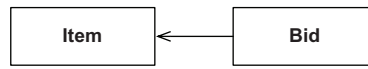


Рис. 7.13 ❖ Отношение между классами `Item` и `Bid`

Позже мы сделаем эту связь двунаправленной, связав `Item` и `Bid` отношением *один ко многим*.

Связь *многие к одному* проще, поэтому начнем с нее. Другие виды связей – *многие ко многим* и *один к одному* – сложнее, поэтому мы рассмотрим их в следующей главе.

Начнем со связи *многие к одному*.

7.3.1. Самая простая связь

Мы называем отображение свойства `Bid#item` *однонаправленной связью многие к одному*. Прежде чем начать обсуждение этого отображения, взгляните на схему базы данных на рис. 7.14.

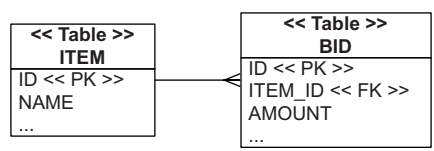


Рис. 7.14 ❖ Отношение *многие к одному* в схеме SQL

Листинг 7.14 ❖ У объекта `Bid` определена единственная ссылка на объект `Item`

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/bidirectional/Bid.java

```
@Entity
public class Bid {

    @ManyToOne(fetch = FetchType.LAZY)  ← По умолчанию – EAGER
    @JoinColumn(name = "ITEM_ID", nullable = false)
    protected Item item;

    // ...
}
```

Аннотация `@ManyToOne` отмечает свойство как связь с сущностью и не является обязательной. К сожалению, параметр `fetch` по умолчанию получает значение `EAGER`: это значит, что вместе с экземпляром `Bid` всегда загружается связанный эк-

экземпляр Item. Как правило, в качестве стратегии по умолчанию мы предпочитаем отложенную загрузку; но об этом мы поговорим в разделе 12.1.1.

Связь *многие к одному* прекрасно отображается в столбец внешнего ключа: ITEM_ID в таблице BID. В JPA это называется *столбцом соединения* (join column). Для ее объявления достаточно только аннотации @ManyToOne. По умолчанию столбец соединения получает имя ITEM_ID: Hibernate автоматически использует комбинацию из имени целевой сущности и ее свойства-идентификатора, разделенных подчеркиванием.

Имя столбца внешнего ключа можно переопределить, используя аннотацию @JoinColumn. Но мы применили ее здесь по иной причине – чтобы при создании схемы SQL фреймворк Hibernate добавил в столбец внешнего ключа ограничение NOT NULL. Ставка всегда должна иметь ссылку на товар – она не может существовать сама по себе (обратите внимание, что это уже указывает на некоторый вид зависимости жизненного цикла, который нужно учитывать). С другой стороны, мы могли отметить эту ассоциацию как обязательную – с помощью @ManyToOne(optional = false) или, как обычно, с помощью аннотации @NotNull из Bean Validation.

Это было довольно просто. Чрезвычайно важно понимать, что вы можете создавать полноценные и сложные приложения, не используя ничего другого.

Отображать другую сторону данного отношения не обязательно; вы можете проигнорировать связь *многие к одному* от Item к Bid. В схеме базы данных есть только столбец внешнего ключа, который вы уже отобразили. Мы говорим серьезно: когда вам встретятся столбец внешнего ключа и два класса, участвующих в отношении сущностей, вы, скорее всего, должны отобразить их с помощью @ManyToOne и ничего более. Теперь можно получить товар (Item) для каждой ставки (Bid), вызвав someBid.getItem(). Реализация JPA разыменует внешний ключ и загрузит экземпляр Item за вас; она также возьмет на себя управление значениями внешних ключей. Но как теперь получить все ставки для товара? Для этого можно написать запрос и выполнить его с помощью объекта EntityManager, используя любой язык запросов, какой поддерживает Hibernate. Например, на JPQL можно написать: select b from Bid b where b.item = :itemParameter. Одна из причин, почему вы выбрали полноценный инструмент ORM, такой как Hibernate, состоит в нежелании создавать и выполнять этот запрос самим.

7.3.2. Определение двунаправленной связи

В начале главы мы привели доводы в пользу отображения коллекции Item#images. Сделаем то же для коллекции Item#bids. Эта коллекция станет реализацией отношения *один ко многим* между классами сущностей Item и Bid. Создав и отобразив поле этой коллекции, вы получите следующее:

- Hibernate автоматически будет выполнять SQL-запрос SELECT * from BID where ITEM_ID = ? при вызове метода someItem.getBids() или в начале обхода элементов коллекции;
- вы сможете: *каскадно* передавать изменения состояния от экземпляра Item ко всем дочерним экземплярам Bid в коллекции; сами выбирать, какие собы-

тия жизненного цикла должны быть транзитивными, например объявить, что все дочерние объекты `Bid` должны сохраняться вместе с родительскими экземплярами `Item`, чтобы не приходилось вызывать метода `EntityManager#persist()` повторно для каждого элемента коллекции.

Что ж, это не очень длинный список. Главным преимуществом отображения *один ко многим* является поддержка доступа к данным во время навигации. Обеспечение доступа к данным только посредством вызова методов предметной модели на Java является одной из основных целей ORM. Как предполагается, за «интеллектуальную» загрузку данных должен отвечать механизм ORM, а ваша задача – работать с собственноручно спроектированными высокоуровневыми интерфейсами: `someItem.getBids().iterator().next().getAmount()` и т. д.

Возможность каскадной передачи некоторых изменений дочерним экземпляром является приятным дополнением. Тем не менее обратите внимание, что некоторые зависимости на уровне Java указывают на типы-значения, а не на сущности. Спросите себя: будет ли хоть одна таблица в схеме иметь столбец внешнего ключа `BID_ID`? Если нет, отобразите класс `Bid` как встраиваемый с аннотацией `@Embeddable`, а не `@Entity`, используя те же таблицы, что и прежде, но с другими отображениями, задающими конкретные правила передачи изменений состояния. Если в какой-нибудь таблице есть внешний ключ, ссылающийся на записи в таблице `BID`, вам потребуется разделяемая сущность `Bid` – в таком случае она не может отображаться как встроенная в `Item`.

Нужно ли вообще отображать коллекцию `Item#bids`? Вы, конечно, получаете доступ к данным при навигации, но за это придется заплатить дополнительным Java-кодом и значительным увеличением сложности. Почти всегда это будет непростой выбор. Как часто вы будете вызывать `someItem.getBids()` в своем приложении, а затем обходить/отображать *все* ставки в предопределенном порядке? Если вам потребуется показать лишь подмножество ставок или извлекать их каждый раз в разном порядке, вам все равно придется писать и выполнять запросы вручную. Отображение *один ко многим* и соответствующая коллекция только усложнят сопровождение. По нашему опыту это часто становится источником проблем и ошибок, особенно среди новичков в ORM.

В случае приложения CaveatEmptor ответ будет положительным: нам часто придется вызывать метод `someItem.getBids()` и передавать список ставок пользователю, желающему участвовать в аукционе. Обновленная диаграмма UML с двунаправленной связью показана на рис. 7.15.

Отображение коллекции и другой стороны связи *один ко многим* выглядит так.

Листинг 7.15 ❖ В классе `Item` объявлена коллекция ссылок на `Bid`

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/bidirectional/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",          ← Требуется для двунаправленной связи
```

```
            fetch = FetchType.LAZY)  ← Значение по умолчанию
    protected Set<Bid> bids = new HashSet<>();
    // ...
}
```

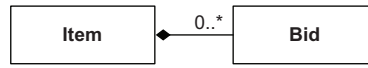


Рис. 7.15 ❖ Двунаправленная связь между классами `Item` и `Bid`

Аннотация `@OneToMany` является обязательной. В данном случае мы также должны задать значение параметра `mappedBy`. Аргументом является имя поля на «другой стороне».

Взгляните еще раз на другую сторону отношения *многие к одному* в листинге 7.15. Свойство в классе `Bid` называется `item`. На стороне ставки (Bid) оно отвечает за столбец внешнего ключа `ITEM_ID`, отображаемого аннотацией `@ManyToOne`. Параметр `mappedBy` сообщает Hibernate, что данную коллекцию нужно загружать, используя столбец внешнего ключа, соответствующий этому полю, в данном случае `Bid#item`. Параметр `mappedBy` обязательно должен определяться для двунаправленных отношений *один ко многим*, если вы уже отобразили столбец внешнего ключа. Мы вернемся к этому обсуждению в следующей главе.

По умолчанию параметр `fetch` отображения коллекции получает значение `FetchType.LAZY`. В будущем этот параметр вам не понадобится. Ему также можно присвоить значение `EAGER`, но оно редко применяется на практике – нечасто требуется, чтобы вместе с экземпляром `Item` загружалась вся коллекция ставок `bids`. Они должны загружаться при обращении, то есть по требованию.

Вторая причина отображения коллекции `Item#bids` – возможность каскадной передачи изменений состояния.

7.3.3. Каскадная передача состояния

Если изменение состояния сущности можно каскадно передать через связь с другой сущностью, для управления отношениями вам потребуется меньше строк кода. Далее показан код, создающий экземпляры `Item` и `Bid` и связывающий их вместе:

```
Item someItem = new Item("Some Item");

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);  ← Не забывайте!
```

Помните об обеих сторонах отношения: конструктор класса `Bid` принимает ссылку на товар, которая присваивается свойству `Bid#item`. Для поддержания целостности экземпляров в памяти вы должны добавить ссылку на ставку в коллекцию `Item#bids`. Теперь, с точки зрения Java-кода, связь установлена, все ссылки присвоены. Если вы не понимаете, зачем нужен этот код, прочитайте раздел 3.2.4.

Давайте сохраним товар со всеми ставками в базу данных – сначала без, а затем с применением механизма транзитивного сохранения.

Транзитивное сохранение

С текущими отображениями `@ManyToOne` и `@OneToMany` для сохранения экземпляра `Item` и нескольких экземпляров `Bid` нам потребуется следующий код.

Листинг 7.16 ❖ Раздельное управление независимыми экземплярами сущностей

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyBidirectional.java

```
Item someItem = new Item("Some Item");
em.persist(someItem);

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);  ← Не забывайте!
em.persist(someBid);

Bid secondBid = new Bid(new BigDecimal("456.00"), someItem);
someItem.getBids().add(secondBid);
em.persist(secondBid);

tx.commit();  ← Проверка изменений; выполнение кода SQL
```

При создании нескольких ставок вызывать метод `persist()` для каждого объекта кажется излишеством. Новые экземпляры являются временными (transient) и должны сохраняться. Отношение между `Bid` и `Item` не влияет на их жизненный цикл. Если бы класс `Bid` был типом-значением, состояние экземпляра `Bid` автоматически соответствовало бы состоянию сущности-владельца `Item`. В нашем случае экземпляр `Bid` обладает собственным, полностью независимым состоянием.

Мы говорили выше, что для представления зависимостей между связанными классами сущностей иногда требуется возможность управления на более низком уровне; это как раз такой случай. Механизмом такого управления в JPA является параметр `cascade`. Например, чтобы сохранить все ставки вместе с товаром, отобразите коллекцию, как показано далее.

Листинг 7.17 ❖ Каскадная передача состояния хранения от экземпляра `Item` каждому элементу коллекции `bids`

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/cascadepersist/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item", cascade = CascadeType.PERSIST)
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Тип каскадной передачи выбирается для операций, которые, по вашему мнению, должны быть транзитивными, поэтому здесь для операции `EntityManager#persist()` используется значение `CascadeType.PERSIST`. Теперь код, связывающий товар со ставками и сохраняющий их, можно упростить.

Листинг 7.18 ❖ Все элементы коллекции `bids` сохраняются автоматически

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyCascadePersist.java

```
Item someItem = new Item("Some Item");
em.persist(someItem);  ← Автоматически сохранит ставку (позже, во время выталкивания контекста)

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);

Bid secondBid = new Bid(new BigDecimal("456.00"), someItem);
someItem.getBids().add(secondBid);

tx.commit();  ← Проверка изменений; выполнение кода SQL
```

В момент подтверждения транзакции Hiberante проверяет все управляемые/хранимые экземпляры `Item`, а также коллекцию `bids`. Затем сам вызывает метод `persist()` для каждого зависимого экземпляра `Bid`, сохраняя их. Значение для столбца `BID#ITEM_ID` извлекается из свойства `Bid#item` каждого объекта `Bid`. Столбец внешнего ключа «отображается» с помощью аннотации `@ManyToOne` над ним.

Аннотация `@ManyToOne` также имеет параметр `cascade`. Но он редко используется на практике. Например, мы же не можем сказать: «когда сохраняется ставка, нужно сохранить и товар». Товар уже должен существовать, иначе ставка не пройдет проверку в базе данных. Подумайте еще об одном возможном отношении `@ManyToOne`: свойстве `Item#seller`, ссылающемся на продавца. Пользователь (`User`) должен существовать раньше, чем он сможет продать товар (`Item`).

Транзитивное сохранение (transitive persistence) – простое понятие, которое часто оказывается полезным при работе с отображениями `@OneToMany` или `@ManyToMany`. Транзитивное удаление (transitive deletion), с другой стороны, нужно применять крайне осторожно.

Каскадное удаление

Кажется разумным вместе с товаром удалять все его ставки, поскольку сами по себе они бессмысленны. Именно это и означает композиция (закрашенный ромбик) на UML-диаграмме. С текущими настройками каскадной обработки мы должны написать следующий код для удаления товара:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyCascadePersist.java

```
Item item = em.find(Item.class, ITEM_ID);

for (Bid bid : item.getBids()) {
    em.remove(bid);  ← ❶ Удаляет ставки
}

em.remove(item);     ← ❷ Удаляет владельца
```

Сначала нужно удалить ставки ❶, а затем и владельца `Item` ❷. Порядок удаления важен. Если сначала удалить объект `Item`, получится нарушение ограничения по внешнему ключу, поскольку операции SQL добавляются в очередь в том же порядке, что и вызовы `remove()`. Поэтому сначала нужно удалить записи из таблицы `BID` и только затем запись из таблицы `ITEM`.

Чтобы облегчить эту работу, JPA предоставляет поддержку каскадирования операций. Механизм хранения может удалять связанные экземпляры сущностей автоматически.

Листинг 7.19 ❖ Каскадное удаление всех элементов коллекции `bids` объекта `Item`

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/cascaderemove/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",
               cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Так же как и раньше для значения `PERSIST`, Hibernate теперь каскадно будет выполнять операцию `remove()` для этого отношения. Если вызвать метод `EntityManager#remove()` объекта `Item`, Hibernate загрузит элементы коллекции `bids` и для каждого экземпляра вызовет метод `remove()`:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyCascadeRemove.java

```
Item item = em.find(Item.class, ITEM_ID);
em.remove(item);  ← После загрузки ставок удаляет их по одной
```

Коллекция должна быть загружена, потому что каждый объект `Bid` – это экземпляр независимой сущности, который должен пройти через обычный жизненный цикл. Если в классе `Bid` есть метод обратного вызова с аннотацией `@PreRemove`, Hibernate должен его вызвать. Больше о состоянии объектов и обратных вызовах вы узнаете в главе 13.

Такой процесс удаления неэффективен: Hibernate должен всегда сначала загрузить коллекцию, а затем удалить каждый объект `Bid` индивидуально. Однако с тем же эффектом можно выполнить единственное SQL-выражение: `delete from BID where ITEM_ID = ?`.

Нам это известно, поскольку в базе данных нет ни одной таблицы с внешним ключом, ссылающимся на таблицу `BID`. Hibernate этого не знает и не может искать по всей базе данных запись, в которой может быть столбец `BID_ID`.

Если бы поле `Item#bids` было коллекцией встраиваемых компонентов, вызов `someItem.getBids().clear()` выполнил бы единственную SQL-операцию `DELETE`. В случае коллекции типов-значений Hibernate полагает, что, скорее всего, никто не может ссылаться на экземпляры ставок, и простое удаление ссылки из коллекции превратит их в «осиротевшие» объекты.

Удаление осиротевших объектов

JPA поддерживает (спорный) флаг, определяющий одинаковое поведение для связей `@OneToMany` (и только `@OneToMany`).

Листинг 7.20 ❖ Удаление осиротевших объектов из коллекции `@OneToMany`

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/orphanremoval/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",
               cascade = CascadeType.PERSIST,
               orphanRemoval = true)  ← Включает CascadeType.REMOVE
    protected Set<Bid> bids = new HashSet<>();
    // ...
}
```

Аргумент `orphanRemoval=true` сообщает фреймворку Hibernate, что тот должен навсегда удалять объекты `Bid` при удалении их из коллекции. Вот пример удаления одного экземпляра `Bid`:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyOrphanRemoval.java

```
Item item = em.find(Item.class, ITEM_ID);
Bid firstBid = item.getBids().iterator().next();
item.getBids().remove(firstBid);  ← Одна ставка удалена
```

Hibernate постоянно отслеживает состояние коллекции, поэтому в момент подтверждения транзакции он обнаружит, что оттуда был удален элемент. Теперь Hibernate будет считать, что объект `Bid` – это осиротевший объект. Вы должны гарантировать, что больше никто не будет на него ссылаться; единственной ссылкой была только что удаленная из коллекции. Hibernate автоматически выполнит SQL-операцию `DELETE` и удалит экземпляр `Bid` из базы данных.

Но метод `clear()` по-прежнему не будет выполняться за одну операцию `DELETE`, как в случае с коллекцией компонентов. Hibernate не нарушает стандартных переходов между состояниями сущности, и каждая ставка загружается и удаляется индивидуально.

Но почему удаление осиротевших объектов считается сомнительной операцией? В данном примере она как раз уместна. Пока в базе данных нет ни одной таб-

лицы, внешний ключ которой ссылался бы на таблицу BID, удаление записи из таблицы BID не вызовет последствий; все ссылки на ставки, хранящиеся в памяти, находятся в коллекции Item#bids. До тех пор, пока выполняются эти условия, удаление осиротевших объектов не представляет проблемы. Это удобно, например, когда уровень представления может изъять элемент из коллекции для удаления чего-нибудь; вы работаете только с экземплярами предметной модели, и вам не нужно вызывать службу для этой операции.

Посмотрим теперь, что произойдет, если определить поле User#bids – другое отображение коллекции @OneToMany, показанное на рис. 7.16. Кстати, это хороший повод проверить ваши знания: как будут выглядеть таблицы и схема после этого изменения? (Ответ: в таблице BID появится столбец внешнего ключа BIDDER_ID, ссылающийся на таблицу USERS.)

Рис. 7.16 ❖ Двунаправленные связи между классами Item, Bid и User

Тест в следующем листинге потерпит неудачу.

Листинг 7.21 ❖ Hibernate не очищает ссылки в памяти после удаления из базы данных

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyOrphanRemoval.java

```
User user = em.find(User.class, USER_ID);
assertEquals(user.getBids().size(), 2);   ← Пользователь сделал две ставки

Item item = em.find(Item.class, ITEM_ID);
Bid firstBid = item.getBids().iterator().next();
item.getBids().remove(firstBid);          ← Одна ставка удалена

// ТЕСТ НЕ ПРОЙДЕТ!
// assertEquals(user.getBids().size(), 1);
assertEquals(user.getBids().size(), 2);   ← Их по-прежнему две!
```

Hibernate считает, что удаленный объект Bid осиротел и его можно удалить; он будет удален из базы данных автоматически, но в коллекции User#bids имеется вторая ссылка на него. После подтверждения этой транзакции база данных останется в корректном состоянии; запись, удаленная из таблицы BID, содержала оба внешних ключа: ITEM_ID и BIDDER_ID. Но в оперативной памяти вы получите противоречивое состояние, поскольку фраза «Удали этот экземпляр при удалении ссылки из коллекции» естественно вызовет конфликт при использовании разделяемых ссылок.

Вместо удаления осиротевших объектов или параметра `CascadeType.REMOVE` всегда ищите более простое отображение. В данном примере поле `Item#bids` можно представить в виде коллекции компонентов, отображаемой аннотацией `@ElementCollection`. Класс `Bid` был бы тогда встраиваемым (`@Embeddable`) и имел бы свойство `bidder` с аннотацией `@ManyToOne`, ссылающееся на экземпляр `User` (встраиваемые компоненты могут определять однонаправленные связи, направленные к сущностям).

Это дало бы требуемый жизненный цикл – полную зависимость от сущности-владельца. Избегайте разделяемых ссылок; на UML-диаграмме (рис. 7.16) связь от класса `Bid` к классу `User` сделана однонаправленной. Избавьтесь от коллекции `User#bids` – это отображение `@OneToMany` не нужно. Если вам нужны все ставки, сделанные пользователем, создайте запрос: `select b from Bid b where b.bidder = :userParameter` (в следующей главе мы дополним это отображение аннотацией `@ManyToOne` во встраиваемом компоненте).

Особенности Hibernate

Использование ON DELETE CASCADE для внешнего ключа

Все продемонстрированные операции удаления неэффективны – каждая ставка должна загружаться в память, и нужно выполнять много SQL-операций `DELETE`. Базы данных SQL оказывают более эффективную поддержку внешнего ключа: параметр `ON DELETE`. Для таблицы `BID` на языке DDL это выглядит так: `foreign key (ITEM_ID) references ITEM on delete cascade`.

Этот параметр говорит базе данных, что нужно поддерживать прозрачную ссылочную целостность для приложения, обращающегося к базе данных. Когда бы вы ни удалили запись из таблицы `ITEM`, база данных автоматически удалит из таблицы `BID` все записи с таким же значением ключа `ITEM_ID`. Для рекурсивного удаления всех зависимых данных потребуется только одна операция `DELETE`, и ничего не придется загружать в память приложения (сервера). Проверьте, включено ли это ограничение для внешних ключей в вашей схеме. Если вы захотите использовать его в схеме, сгенерированной Hibernate, используйте Hibernate-аннотацию `@OnDelete`.

Листинг 7.22 ❖ Генерация схемы с ограничением внешнего ключа `ON DELETE CASCADE`

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/ondeletecascade/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item", cascade = CascadeType.PERSIST)
    @org.hibernate.annotations.OnDelete(
        action = org.hibernate.annotations.OnDeleteAction.CASCADE
    )
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Особенность Hibernate: параметры схемы обычно задаются на «другой» стороне связи, на которую указывает параметр «mappedBy»

Мы видим здесь одну из особенностей Hibernate: аннотация `@OnDelete` влияет только на схему, которую генерирует Hibernate. Настройки, влияющие на формирование схемы, обычно находятся на «другой» стороне `mappedBy` – там, где определяется отображение внешнего ключа/столбца соединения. Соответствующая аннотация `@OnDelete` обычно помещается рядом с `@ManyToOne` в классе `Bid`. Но если связь двунаправленная, Hibernate распознает ее только на стороне `@OneToMany`.

Применение ограничения каскадного удаления для внешнего ключа в базе данных не влияет на поведение Hibernate во время выполнения. Вы можете столкнуться с теми же проблемами, что и в листинге 7.21. Данные в памяти могут не отражать состояния базы данных. Если все записи в таблице `BID` автоматически удаляются при удалении записи из таблицы `ITEM`, тогда код приложения должен очищать ссылки и синхронизировать данные в памяти с содержимым в базе данных. По неосторожности можно заново сохранить то, что было удалено до этого.

Экземпляры `Bid` не следуют стандартному жизненному циклу, и такие методы обратных вызовов, как `@PreRemove`, не окажут никакого эффекта. Кроме того, Hibernate не очищает необязательный глобальный кэш второго уровня автоматически, что может привести к устареванию данных. По сути, все проблемы, с которыми вы можете столкнуться, используя каскадные изменения для внешнего ключа на уровне базы данных, такие же, как в случае, когда другое приложение, кроме вашего, обращается к той же базе данных либо когда изменения вносит триггер базы данных. Hibernate может быть очень эффективным инструментом в подобном случае, но не нужно забывать про остальные движущиеся части системы. Мы подробнее рассмотрим многопоточность и кэширование позже в этой книге.

При разработке новой схемы проще не применять каскадных изменений на уровне базы данных и отобразить отношение композиции в предметной модели как встроенное/встраиваемое, а не как связь с сущностью. В этом случае Hibernate сможет выполнять эффективные SQL-операции `DELETE` для удаления композиций целиком. Повторим рекомендацию из предыдущего раздела: если можно избежать использования общих ссылок, отображайте класс `Big` как `@ElementCollection` в классе `Item`, а не как отдельную сущность со связями @ManyToOne и `@OneToMany`. Конечно, можно вообще не отображать никаких коллекций, используя простейшее отображение: столбец внешнего ключа с аннотацией `@ManyToOne` и однонаправленную связь между классами сущностей (`@Entity`).

7.4. Резюме

- Вы познакомились со множеством интерфейсов и реализаций, использующих простые отображения коллекций, такие как `Set<String>`.
- Вы узнали, как работают отсортированные коллекции, а также средства, применяемые в Hibernate для получения элементов из базы данных в заданном порядке.
- Мы обсудили сложные коллекции пользовательских встраиваемых типов, а также множества, контейнеры и словари компонентов.

- Вы увидели, как использовать компоненты в качестве ключей и значений в словарях и как определять коллекции во встраиваемых компонентах.
- Отображение первого столбца внешнего ключа в связь с сущностью *многие к одному* делает ее двунаправленной, так же как связь *один ко многим*. Вы также узнали о возможности каскадной передачи изменений.
- Мы обсудили основные понятия объектно-реляционного отображения. После того как вы отобразите первую связь `@ManyToOne` и, возможно, коллекцию строк, худшее окажется позади.
- Обязательно выполните код примеров (и загляните в журнал SQL)!

Глава 8

Продвинутые приемы отображения связей между сущностями

В этой главе:

- отображение связей вида *один к одному*;
- варианты отображения связи *один ко многим*;
- связь *многие ко многим* и тройные отношения между сущностями.

В предыдущей главе мы рассмотрели однонаправленную связь *многие к одному*, сделали ее двунаправленной, а затем реализовали транзитивную передачу состояния посредством каскадирования. Причина, по которой мы обсуждаем более продвинутые приемы отображения сущностей в отдельной главе, заключается в том, что многие из них редко находят практическое применение и почти всегда есть возможность не использовать их. Обычно можно обойтись лишь отображениями компонентов и связями между сущностями *многие к одному* (иногда *один к одному*). Можно даже написать довольно сложное приложение, не отобразив ни одной коллекции! В предыдущей главе мы перечислили определенные преимущества отображения коллекций; эти же соображения относятся ко всем примерам данной главы. Прежде чем начать реализацию сложного отображения коллекции, всегда нужно убедиться в ее необходимости.

Начнем с отображения, в котором коллекции не задействованы, – связи сущностей вида *один к одному*.

Главные нововведения в JPA 2

- Связи *многие к одному* и *один к одному* теперь могут отображаться с помощью промежуточной таблицы соединения.
- Классы встраиваемых компонентов могут иметь однонаправленные связи с сущностями, даже когда они содержат коллекции компонентов.

8.1. Связи *один к одному*

В разделе 5.2 мы доказывали, что отношение между классами `User` и `Address` – когда пользователь имеет платежный адрес (`billingAddress`), домашний адрес (`homeAddress`) и адрес доставки (`shippingAddress`) – лучше представить в виде отображения встраиваемого (`@Embeddable`) компонента. Как правило, это простейший способ представления отношений *один к одному*, поскольку в подобных случаях реализуется зависимый жизненный цикл компонентов. На UML-диаграмме это представлено либо агрегацией, либо композицией.

Но что, если создать отдельную таблицу `ADDRESS`, отображая классы `User` и `Address` как отдельные сущности? Одним из преимуществ такого подхода является возможность разделения ссылок: другая сущность, например `Shipment` (доставка), сможет ссылаться на конкретный экземпляр класса `Address`. Если поле `shippingAddress` объекта `User` будет ссылаться на этот же адрес, экземпляр `Address` должен обладать собственной идентичностью для поддержки общих ссылок.

В таком случае между классами `User` и `Address` возникнет настоящая связь *один к одному*. Взгляните на исправленную диаграмму классов на рис. 8.1.

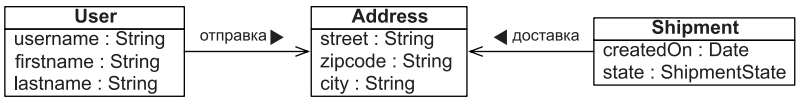


Рис. 8.1 ❖ Класс `Address` как сущность с двумя связями, допускающая разделение ссылок

Существует несколько способов отображения связи *один к одному*. Первая стратегия, которую мы рассмотрим, заключается в использовании общего значения первичного ключа.

8.1.1. Общий первичный ключ

Записи в двух таблицах, связанные по первичному ключу, имеют одинаковые значения первичных ключей. У пользователя (`User`) такой же первичный ключ, как и у его адреса (доставки), объекта `Address`. Самое сложное в данном подходе – гарантировать присваивание связанным экземплярам одинакового значения первичного ключа. Прежде чем исследовать эту проблему, создадим основное отображение. Класс `Address` – теперь самостоятельная сущность: он больше не является компонентом.

Листинг 8.1 ❖ Класс `Address` как самостоятельная сущность

Файл: /model/src/main/java/org/jpwh/model/associations/onetoone/sharedprimarykey/Address.java

```
@Entity
public class Address {

    @Id
```

```
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String street;

    @NotNull
    protected String zipcode;

    @NotNull
    protected String city;

    // ...
}
```

Класс `User` также является сущностью, связь в которой представлена полем `shippingAddress`.

Листинг 8.2 ❖ Сущность `User` и ассоциация `shippingAddress`

Файл: /model/src/main/java/org/jpwh/model/associations/onetoone/sharedprimarykey/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @Id  <-- ❶ По умолчанию EAGER
    protected Long id;

    @OneToOne(  <-- ❷ Значение идентификатора присваивается приложением
        fetch = FetchType.LAZY,
        optional = false
    )
    @PrimaryKeyJoinColumn  <-- ❺ Выбор стратегии общего первичного ключа
    protected Address shippingAddress;

    protected User() {
    }

    public User(Long id, String username) {  <-- ❻ Требуется идентификатор
        this.id = id;
        this.username = username;
    }
    // ...
}
```

❸ Помечает поле с типом сущности как ассоциацию *один к одному*

❹ Обязательно для поддержки отложенной загрузки с помощью прокси-объектов

В классе `User` не нужно объявлять автоматическое создание идентификаторов ❶. Как было сказано в разделе 4.2.4, это один из немногих случаев, когда значение идентификатора *выбирается приложением*. В строке ❻ видно, как конструктор гарантирует это условие: публичный API требует указать значение идентификатора при создании экземпляра.

В этом примере появились две новые аннотации. Аннотация `@OneToOne` ❷, как вы наверняка догадались, отмечает свойство с типом сущности как связь *один*

к одному. Как обычно, желательно применять стратегию отложенной загрузки, поэтому значение по умолчанию `FetchType.EAGER` заменяется на `LAZY` ❸. Вторая новая аннотация – `@PrimaryKeyJoinColumn` ❺ – определяет требуемую стратегию отображения с общим первичным ключом. В результате получается отображение однонаправленной связи *один к одному* с общим первичным ключом, направленной от класса `User` к классу `Address`.

Параметр `optional=false` ❹ указывает, что объект `User` должен иметь непустую ссылку `shippingAddress`. Фреймворк Hibernate отразит этот факт в сгенерированной схеме базы данных с помощью ограничения внешнего ключа. Первичный ключ таблицы `USERS` также является внешним ключом, ссылающимся на первичный ключ таблицы `ADDRESS`. См. рис. 8.2.

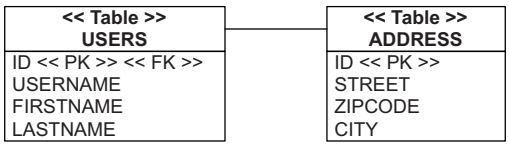


Рис. 8.2 ❖ Первичный ключ таблицы `USERS` отдновременно является внешним ключом

Спецификация JPA не предоставляет стандартных методов создания общих первичных ключей. А это значит, что только вы отвечаете за присваивание экземпляру `User` такого же значения идентификатора, как у связанного объекта `Address`, перед сохранением:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToOneSharedPrimaryKey.java

```
Address someAddress =
    new Address("Some Street 123", "12345", "Some City");

em.persist(someAddress);  ← Формирует значение идентификатора

User someUser =
    new User(
        someAddress.getId(),  ← Присваивает такое же значение идентификатора
        "johndoe"
    );

em.persist(someUser);

someUser.setShippingAddress(someAddress);  ← Необязательно
```

После сохранения объекта `Address` нужно взять его идентификатор и присвоить его экземпляру `User` перед сохранением. Последняя строка в этом примере не является обязательной, но поскольку код ожидает получить значение при вызове `someUser.getShippingAddress()`, вы должны присвоить свойству значение. Hibernate не сообщит об ошибке, если вы опустите этот последний шаг.

Данный способ отображения и его реализация имеют три проблемы:

- вы должны помнить, что объект `Address` нужно сохранять первым, а затем, после вызова `persist()`, извлечь из него значение идентификатора. Это возможно лишь в случае, когда в классе `Address` объявлен генератор идентификаторов, формирующий значения во время вызова `persist()`, но перед выполнением `INSERT`, как обсуждалось в разделе 4.2.5. Иначе вызов `someAddress.getId()` вернет `null` и вы не сможете вручную установить значение идентификатора в объекте `User`;
- отложенная загрузка с помощью прокси-объектов будет работать, только если связь является обязательной. Обычно это становится неожиданностью для разработчиков, только начинающих работу с JPA. Для аннотации `@OneToOne` по умолчанию используется режим загрузки `FetchType.EAGER`: при загрузке экземпляра `User` Hibernate сразу же загрузит и `shippingAddress`. Концептуально отложенная загрузка с помощью прокси-объектов имеет смысл, лишь когда Hibernate знает о существовании объекта, связанного со свойством `shippingAddress`. Если свойство может содержать null, Hibernate должен проверять, не содержит ли поле в базе данных значения `NULL`, выполняя запрос к таблице `ADDRESS`. Но если приходится проверять состояние базы данных, можно сразу же загрузить значение, поскольку никакого выигрыша от использования прокси-объекта не будет;
- связь *один к одному* – однонаправленная; иногда может потребоваться реализовать возможность двунаправленной навигации.

Первая проблема не имеет иного решения, и это одна из причин, почему следует использовать генератор идентификаторов, способный формировать значения перед выполнением SQL-операции `INSERT`.

Ассоциация `@OneToOne(optional=true)` не поддерживает отложенную загрузку данных с помощью прокси-объектов. Это соответствует спецификации JPA. Аргумент аннотации `FetchType.LAZY` для реализации механизма хранения является лишь просьбой, но не требованием. Можно обеспечить отложенную загрузку поля `@OneToOne`, которое может содержать null, используя прием внедрения в байт-код, как будет показано в разделе 12.1.3.

Что касается последней проблемы, сделать связь двунаправленной можно, используя оригинальный генератор идентификаторов Hibernate, помогающий присваивать значения первичных ключей.

8.1.2. Генератор внешнего первичного ключа

Двунаправленное отображение всегда требует наличия отображаемой (`mappedBy`) стороны. Здесь мы выберем сторону класса `User` (это дело вкуса и, возможно, каких-то несущественных требований):

Файл: /model/src/main/java/org/jpwh/model/associations/onetoone/foreigngenerator/User.java

```
@Entity
@Table(name = "USERS")
public class User {
```

```
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @OneToOne(
        mappedBy = "user",
        cascade = CascadeType.PERSIST
    )
    protected Address shippingAddress;
    // ...
}
```

Сравните это отображение с предыдущим: здесь мы добавили параметр `mappedBy`, сообщив Hibernate, что низкоуровневые детали теперь отображаются с помощью свойства `user` на «другой стороне». Для удобства укажем параметр `CascadeType.PERSIST`; механизм транзитивного сохранения упростит запись экземпляров в базу данных в правильном порядке. При сохранении экземпляра `User` Hibernate автоматически сохранит объект, на который ссылается свойство `shippingAddress`, а также автоматически сформирует и присвоит идентификатор первичного ключа.

Теперь посмотрим на «другую сторону» – класс `Address`.

Листинг 8.3 ❖ В классе `Address` определен особый генератор значений внешнего ключа

Файл: /model/src/main/java/org/jpwh/model/associations/onetoone/foreigngenerator/Address.java

```
@Entity
public class Address {

    @Id
    @GeneratedValue(generator = "addressKeyGenerator")
    @org.hibernate.annotations.GenericGenerator(
        name = "addressKeyGenerator",
        strategy = "foreign",
        parameters =
            @org.hibernate.annotations.Parameter(
                name = "property", value = "user"
            )
    )
    protected Long id;

    @OneToOne(optional = false)
    @PrimaryKeyJoinColumn
    protected User user;

    protected Address() {
    }

    public Address(User user) {
        this.user = user;
    }

    public Address(User user, String street, String zipcode, String city) {
```

❶ Определяет генератор значений первичного ключа

❷ Создает ограничение внешнего ключа

❸ Адрес должен ссылаться на пользователя

❹ Общедоступные конструкторы класса Address

```
        this.user = user;
        this.street = street;
        this.zipcode = zipcode;
        this.city = city;
    }
    // ...
}
```

Здесь довольно много нового кода. Сначала рассмотрим свойство идентификатора, а затем связь *один к одному*.

Особенности Hibernate

Аннотация `@GenericGenerator` перед свойством идентификатора ❶ определяет специальный генератор значений первичного ключа, использующий стратегию `foreign`, доступную только в Hibernate. Мы не рассматривали этого генератора в разделе 4.2.5; его единственное применение – связь *один к одному* с общим первичным ключом. При сохранении экземпляра `Address` этот специальный генератор извлекает значение идентификатора из экземпляра сущности `User`, на который ссылается свойство `user`.

Далее рассмотрим отображение `@OneToOne` ❷. С помощью аннотации `@PrimaryKeyJoinColumn` ❸ свойство `user` отмечено как связь между сущностями с общим первичным ключом. Параметр `optional=false` в аннотации требует, чтобы объект `Address` имел ссылку на объект `User`. Теперь в вызовы общедоступных конструкторов `Address` ❹ требуется передавать экземпляр `User`. Благодаря параметру `optional=false` столбец первичного ключа в таблице `ADDRESS` теперь имеет ограничение внешнего ключа, как показано в схеме на рис. 8.3.

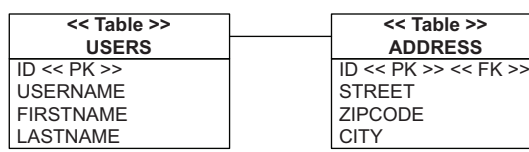


Рис. 8.3 ❖ Ограничение внешнего ключа для столбца первичного ключа таблицы `ADDRESS`

Больше не нужно вызывать `someAddress.getId()` или `someUser.getId()` при выполнении единицы работы. Теперь сохранять данные стало проще:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToOneForeignGenerator.java

```
User someUser = new User("johndoe");

Address someAddress =
    new Address(
```

```
        someUser,
        "Some Street 123", "12345", "Some City"
    );
someUser.setShippingAddress(someAddress);

em.persist(someUser);
```

Связывание

Транзитивное сохранение для поля shippingAddress

Не забывайте, что связать нужно обе стороны двунаправленного отношения сущностей. Обратите внимание, что такое отображение дает отложенную загрузку поля `User#shippingAddress` (оно необязательно/может содержать null), но вы сможете загружать поле `Address#user` по требованию (оно обязательно) с помощью прокси-объектов.

Общий первичный ключ в связи *один к одному* используется относительно редко. Вместо этого связи «к одному» обычно отображаются с помощью столбца внешнего ключа с ограничением уникальности.

8.1.3. Соединение с помощью столбца внешнего ключа

Две записи могут быть связаны с помощью дополнительного столбца внешнего ключа вместо общего первичного ключа. В этом случае одна из таблиц должна иметь столбец внешнего ключа, ссылающегося на первичный ключ ассоциированной таблицы. (Источником и целью этого внешнего ключа может выступать одна таблица: мы называем это *рекурсивным отношением*.)

Давайте изменим отображение поля `User#shippingAddress`. Вместо общего первичного ключа добавим в таблицу `USERS` столбец `SHIPPINGADDRESS_ID`. Добавим также ограничение `UNIQUE`, чтобы два пользователя не могли иметь одинакового адреса доставки. Взгляните на схему, показанную на рис. 8.4.

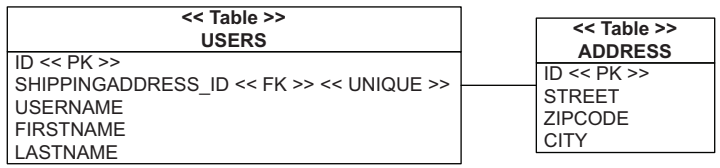


Рис. 8.4 ❖ Столбец соединения для связи *один к одному* между таблицами `USERS` и `ADDRESS`

`Address` – это обычный класс сущности, такой же, как показанный в начале этой главы в листинге 8.1. В классе сущности `User` объявлено поле `shippingAddress`, реализующее однонаправленную связь:

Файл: /model/src/main/java/org/jpwh/model/associations/onetoone/foreignkey/User.java

```
@Entity
@Table(name = "USERS")
```

```
public class User {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @OneToOne(
        fetch = FetchType.LAZY,
        optional = false,        ← NOT NULL
        cascade = CascadeType.PERSIST
    )
    @JoinColumn(unique = true)  ← По умолчанию SHIPPINGADDRESS_ID
    protected Address shippingAddress;

    // ...
}
```

Вам не нужны ни специальные генераторы идентификаторов, ни присваивание значения первичного ключа; вместо аннотации `@PrimaryKeyJoinColumn` пользуйтесь `@JoinColumn` как обычно. Если вы больше знакомы с SQL, нежели с JPA, воспринимайте отображение `@JoinColumn` как столбец внешнего ключа.

Для данной связи требуется использовать отложенную загрузку. В отличие от стратегии с общим первичным ключом, здесь не возникнет проблем с отложенной загрузкой: после загрузки записи из таблицы `USERS` в ней будет содержаться значение столбца `SHIPPINGADDRESS_ID`. Благодаря этому Hibernate узнает о существовании записи в таблице `ADDRESS` и для загрузки экземпляра `Address` по требованию сможет использовать прокси-объект.

Тем не менее в отображении указан параметр `optional=false`, требующий наличия у пользователя адреса доставки. Это не влияет на алгоритм загрузки, но является лишь логическим следствием параметра `unique=true` в аннотации `@JoinColumn`. Этот параметр добавляет ограничение уникальности в сформированную схему SQL. Так как значения столбца `SHIPPINGADDRESS_ID` должны быть уникальны, лишь у одного пользователя адрес доставки может оказаться пустым. Следовательно, нет смысла создавать столбцы, которые могут содержать null и иметь ограничение уникальности.

Создавать, связывать и сохранять экземпляры очень просто:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToOneForeignKey.java

```
User someUser =
    new User("johndoe");

Address someAddress =
    new Address("Some Street 123", "12345", "Some City");

someUser.setShippingAddress(someAddress);  ← Связывание

em.persist(someUser);  ← Транзитивное сохранение поля shippingAddress
```

Вы познакомились с двумя основными способами отображения связи *один к одному*: первый – с общим первичным ключом, второй – с внешним ключом и ограничением уникальности столбца. Последний способ отображения, который мы хотим продемонстрировать, более экзотический: отображение связи *один к одному* с помощью дополнительной таблицы.

8.1.4. Использование таблицы соединения

Вы, наверное, замечали, что столбцы, которые могут содержать null, порой создают проблемы. Иногда лучшим решением для работы с необязательными значениями является промежуточная таблица: если в таблице нашлась строка, значит, ссылка существует, и наоборот.

Давайте рассмотрим сущность `Shipment` в приложении CaveatEmptor и обсудим ее назначение. Продавцы и покупатели взаимодействуют в приложении CaveatEmptor, запуская аукционы и делая на них ставки. Отправка товаров как будто не входит в сферу ответственности приложения; после окончания аукциона продавец и покупатель сами договариваются о способах платежа и доставки. Они могут сделать это и без приложения CaveatEmptor.

С другой стороны, в приложении CaveatEmptor можно было бы предложить услугу условного депонирования (escrow). Продавцы будут использовать ее для создания отслеживаемых поставок по окончании аукциона. Продавец будет оплачивать стоимость аукционного товара доверенному лицу (вам), а вы будете информировать покупателя о поступлении денег. После доставки и приемки товара покупателем вы будете переводить деньги продавцу.

Если вы хоть раз принимали участие в онлайн-аукционе со значительной суммой, то наверняка пользовались подобной услугой. Но от приложения CaveatEmptor хотелось бы большего: не только предоставлять такую услугу для завершившихся аукционов, но также дать пользователям возможность создавать отслеживаемые и надежные отправки для любой сделки, заключенной вне аукциона, вне приложения CaveatEmptor.

Такой сценарий требует наличия сущности `Shipment` с необязательной связью *один к одному* с классом `Item`. Взгляните на диаграмму классов для этой предметной модели на рис. 8.5.

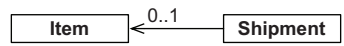


Рис. 8.5 ❖ Отправка (`Shipment`) может быть связана с аукционным товаром (`Item`)

ПРИМЕЧАНИЕ В этом разделе мы сначала хотели отойти от примера с CaveatEmptor, поскольку не смогли найти естественного сценария, который бы требовал необязательной связи *один к одному*. Если этот пример с условным депонированием кажется притянутым за уши, попробуйте рассмотреть проблему распределения сотрудников по рабочим местам. Это такое же необязательное отношение *один к одному*.

В схему базы данных следует добавить промежуточную таблицу связи с именем ITEM_SHIPMENT. Запись в этой таблице представляет объект Shipment, созданный в рамках аукциона. Таблицы показаны на рис. 8.6.

Рис. 8.6 ❖ Промежуточная таблица связывает товар с отправкой

Обратите внимание, как схема гарантирует уникальность и обеспечивает отношение *один к одному*: первичный ключ таблицы ITEM_SHIPMENT хранится в столбце SHIPMENT_ID, а ITEM_ID содержит только уникальные значения. То есть одному товару соответствует лишь одна отправка. Конечно, это также означает, что отправка может содержать лишь один товар.

Такая модель отображается при помощи аннотации @OneToOne в классе сущности Shipment:

Файл: /model/src/main/java/org/jpwh/model/associations/onetoone/jointable/Shipment.java

```
@Entity
public class Shipment {

    @OneToOne(fetch = FetchType.LAZY)
    @JoinTable(
        name = "ITEM_SHIPMENT",  <-- Имя обязательно!
        joinColumns =
            @JoinColumn(name = "SHIPMENT_ID"),  <-- По умолчанию ID
        inverseJoinColumns =
            @JoinColumn(name = "ITEM_ID",  <-- По умолчанию AUCTION_ID
                                    nullable = false,
                                    unique = true)
    )
    protected Item auction;

    public Shipment() {
    }

    public Shipment(Item auction) {
        this.auction = auction;
    }

    // ...
}
```

Отложенная загрузка включается за счет следующей особенности: когда Hibernate загружает экземпляр Shipment, он выполняет запросы к таблице SHIPMENT и таблице соединения ITEM_SHIPMENT. Прежде чем использовать прокси-объект, Hibernate должен убедиться, что ссылка на экземпляр Item существует. Он делает это

при помощи SQL-запроса с внешним соединением, поэтому дополнительных выражений SQL вы не встретите. Если искомая запись имеется в таблице `ITEM_SHIPMENT`, Hibernate вернет прокси-объект для экземпляра `Item`.

Аннотация `@JoinTable` не встречалась нам ранее; она всегда требует указывать имя промежуточной таблицы. Это отображение фактически скрывает таблицу соединения; не существует Java-класса, который бы ей соответствовал. Эта аннотация задает имена столбцов таблицы `ITEM_SHIPMENT`, а Hibernate сгенерирует в схеме ограничение `UNIQUE` для столбца `ITEM_ID`. Также Hibernate сгенерирует подходящие ограничения внешнего ключа для столбцов таблицы соединения.

Следующий фрагмент сохраняет одну отправку (`Shipment`) без товара (`Item`), а другую связывает с одним товаром (`Item`):

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToOneJoinTable.java

```
Shipment someShipment = new Shipment();
em.persist(someShipment);

Item someItem = new Item("Some Item");
em.persist(someItem);

Shipment auctionShipment = new Shipment(someItem);
em.persist(auctionShipment);
```

На этом мы завершим обзор способов отображения связи *один к одному*. В итоге, когда одна из двух сущностей всегда сохраняется раньше другой и может быть источником значений первичного ключа, мы советуем использовать связь с общим первичным ключом. В остальных ситуациях применяйте связь с внешним ключом, а если связь *один к одному* является необязательной, используйте скрытую промежуточную таблицу соединения.

Далее мы сосредоточим внимание на множественных (*many-valued*) связях между сущностями, начиная с некоторых продвинутых способов отображения связи *один ко многим*.

8.2. Связь *один ко многим*

Множественная связь между сущностями – это по определению коллекция ссылок на сущности. Вы уже видели, как отображается связь *один ко многим* в предыдущей главе, в разделе 7.3.2. Связи *один ко многим* представляют наиболее важный тип связей, использующий коллекции. Мы даже рискнем отговорить вас от использования более сложных типов связей, поскольку подойдет и простая двунаправленная ассоциация *многие к одному/один ко многим*.

Помните также, что необязательно отображать все коллекции сущностей, если в этом нет необходимости; всегда можно написать явный запрос вместо прямого обращения к данным во время обхода. Если вы все же решите отобразить коллекцию ссылок на сущности, у вас на выбор есть несколько вариантов, и сейчас мы рассмотрим некоторые наиболее сложные ситуации.

8.2.1. Применение контейнеров в связях *один ко многим*

До сих пор мы использовали связь `@OneToMany` только со множеством (`Set`); однако вместо двунаправленной связи *один ко многим* можно использовать отображение контейнера. Но зачем это делать?

Среди всех коллекций, пригодных для реализации двунаправленной связи *один ко многим*, контейнеры обладают лучшими показателями производительности. По умолчанию коллекции в Hibernate загружаются во время первого обращения к ним. Поскольку контейнер не должен хранить индексов своих элементов (как список) или проверять дублирования элементов (как множество), вы можете добавлять в контейнер новые элементы, не инициируя при этом загрузку данных. Это может быть важно при отображении потенциально большой коллекции ссылок на сущности.

С другой стороны, невозможно одновременно загрузить два контейнера: например, если бы поля `bids` и `images` класса `Item` были контейнерами, связанными отношением *один ко многим*. Это не большая потеря, поскольку одновременное извлечение двух коллекций, как правило, создает декартово произведение; таких операций нужно избегать, независимо от того, используете вы контейнеры, списки или множества. Но отложим пока стратегии извлечения до главы 12. В целом мы считаем, что контейнер является лучшей коллекцией для обратной стороны связи *один ко многим*, когда та отображается аннотацией `@OneToMany(mappedBy = "...")`.

Для отображения двунаправленной связи *один ко многим* с помощью контейнера следует поменять тип коллекции `bids` в классе `Item` на `Collection`, указав тип `ArrayList` в качестве реализации. Отображение связи между классами `Item` и `Bid`, по сути, не поменяется:

Файл: /model/src/main/java/org/jpwh/model/associations/oneto-many/bag/Item.java

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item")
    public Collection<Bid> bids = new ArrayList<>();

    // ...
}
```

Класс `Bid` с аннотацией `@ManyToOne` (отображающая сторона, владелец отношения) и даже таблицы останутся такими же, как в разделе 7.3.1.

В отличие от множества, которое мы отображали ранее, контейнер допускает дублирование элементов:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyBag.java

```
Item someItem = new Item("Some Item");
em.persist(someItem);

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);
```

```
someItem.getBids().add(someBid);  ← Не вызывает немедленого сохранения!
em.persist(someBid);

assertEquals(someItem.getBids().size(), 2);
```

Как оказывается, для данного примера это не актуально, поскольку *дублирование* означает повторное добавление определенной ссылки на экземпляр `Bid`. Вы бы не стали так делать в настоящем приложении. Даже если добавить в коллекцию одну ссылку несколько раз, Hibernate проигнорирует ее. Стороной, влияющей на обновление базы данных, является сторона `@ManyToOne`, которая уже отображает данное отношение. При загрузке экземпляра `Item` в коллекции уже не будет дубликата:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyBag.java

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(item.getBids().size(), 1);
```

Как уже упоминалось, преимущество контейнера заключается в том, что для добавления элемента не требуется инициализировать коллекцию:

Файл: /examples/src/test/java/org/jpwh/test/associations/OneToManyBag.java

```
Item item = em.find(Item.class, ITEM_ID);

Bid bid = new Bid(new BigDecimal("456.00"), item);
item.getBids().add(bid);  ← Никакого SELECT!
em.persist(bid);
```

Этот код вызовет лишь одну SQL-операцию `SELECT` для загрузки экземпляра `Item`. Если вызвать `em.getReference()` вместо `em.find()`, Hibernate все равно инициализирует и вернет прокси-объект `Item`, используя операцию `SELECT`, как только вы вызовете `item.getBids()`. Но до тех пор, пока не начат обход коллекции (`Collection`), не нужны никакие запросы, а операция `INSERT` для новых объектов `Bid` будет выполняться без загрузки поля `bids`. Если коллекция имеет тип `Set` или `List`, Hibernate загрузит все элементы при добавлении нового элемента.

Давайте поменяем тип коллекции, превратив ее в хранимый список (`List`).

8.2.2. Однонаправленное и двунаправленное отображения списка

Если вам потребуется настоящий список, чтобы сохранить позиции элементов в коллекции, вам понадобится дополнительный столбец для хранения индексов. В случае связи *один ко многим* это также означает, что необходимо поменять тип поля `Item#bids` на `List`, инициализируя его объектом `ArrayList`:

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/ list/ Item.java

```
@Entity
public class Item {

    @OneToMany
```

```
    @JoinColumn(
        name = "ITEM_ID",
        nullable = false
    )
    @OrderColumn(
        name = "BID_POSITION",  ← По умолчанию BIDS_ORDER
        nullable = false
    )
    public List<Bid> bids = new ArrayList<>();

    // ...
}
```

Это однонаправленное отображение – другая, отображающая сторона отсутствует. В классе `Bid` отсутствует поле с аннотацией `@ManyToOne`. Для хранения индексов хранимого списка используется новая аннотация `@OrderColumn`, в которой, как обычно, нужно задать для столбца ограничение `NOT NULL`. Представление таблицы `BID` в базе данных со столбцами соединения и упорядочивания показано на рис. 8.7.

BID

ID	ITEM_ID	BID_POSITION	AMOUNT
1	1	0	99.00
2	1	1	100.00
3	1	2	101.00
4	2	0	4.99

Рис. 8.7 ❖ Таблица `BID`

Для каждой коллекции сохраняемый индекс начинается с нуля и является непрерывным (разрывы отсутствуют). Потенциально Hibernate может выполнять множество SQL-операций при добавлении, удалении и сдвиге элементов списка (`List`). Мы уже говорили об этой проблеме производительности в разделе 7.1.6.

Давайте сделаем это отображение двунаправленным с помощью поля `@ManyToOne` сущности `Bid`:

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/list/Bid.java

```
@Entity
public class Bid {

    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        updatable = false, insertable = false  ← Запретить запись!
    )
```

```
    @NotNull  ← Для формирования схемы
    protected Item item;

    // ...
}
```

Вы, наверное, ожидали увидеть другой код, например `@ManyToOne(mappedBy="bids")` без дополнительной аннотации `@JoinColumn`. Но в аннотации `@ManyToOne` отсутствует атрибут `mappedBy`: она всегда находится на стороне, владеющей отношением. Вы должны сделать отображаемой другую сторону с аннотацией `@OneToMany`. Здесь вы столкнетесь с одной концептуальной проблемой и некоторыми нюансами Hibernate.

Коллекция `Item#bids` теперь доступна не только для чтения, поскольку Hibernate должен сохранять индекс каждого элемента. Если бы отношением владела сторона `Bid#item`, Hibernate проигнорировал бы коллекцию при сохранении данных, не записав индексов элементов. Нужно дважды отобразить `@JoinColumn`, а затем запретить запись на стороне `@ManyToOne` с помощью параметров `updatable=false` и `insertable=false`. После этого Hibernate будет учитывать сторону коллекции при сохранении данных, включая индекс каждого элемента. Связь `@ManyToOne` доступна, по сути, только для чтения, как если бы она обладала атрибутом `mappedBy`.

Двунаправленный список с атрибутом mappedBy

В Hibernate есть несколько нерешенных проблем, связанных с этим обстоятельством; будущие версии, возможно, позволят применять к коллекции, совместимые с JPA аннотации `@OneToMany(mappedBy)` и `@OrderColumn`. На момент написания книги представленное отображение было единственной рабочей реализацией двунаправленной связи *один ко многим*, использующей хранимый список (`List`).

Наконец, генератор схемы в Hibernate всегда полагается на аннотацию `@JoinColumn` на стороне `@ManyToOne`. Следовательно, чтобы получить корректную схему, необходимо поместить аннотацию `@NotNull` на этой стороне или применить `@JoinColumn(nullable=false)`. Генератор проигнорирует сторону `@OneToMany` и ее столбец соединения, если обнаружит аннотацию `@ManyToOne`.

В действующем приложении вы бы не стали отображать ассоциацию, используя коллекцию `List`. Сохранение порядка элементов в базе данных кажется распространенным вариантом использования, но, если подумать, он не очень полезен: иногда требуется показывать пользователю список, где сначала идут последние ставки, или ставки, сделанные конкретным пользователем, или ставки, сделанные в заданном временном интервале. Ни одной из этих операций индекс элемента в списке не нужен. Как уже упоминалось в разделе 3.2.4, не следует хранить порядок отображения в базе данных; используйте гибкий подход с применением запросов, а не жестко закодированные отображения. Кроме того, сопровождение индексов при удалении, добавлении или сдвиге элементов списка может быть дорогостоящим и вызывать выполнение множества выражений SQL. Отобразите

столбец соединения с внешним ключом, используя аннотацию `@ManyToOne`, и избавьтесь от коллекции.

Далее будет представлен еще один сценарий с отношением *один ко многим*: с отображением связи в промежуточную таблицу соединения.

8.2.3. Необязательная связь *один ко многим* с таблицей соединения

Полезным дополнением к классу `Item` может стать свойство `buyer` (покупатель). Благодаря ему вы сможете вызвать метод `someItem.getBuyer()`, чтобы получить пользователя (`User`), сделавшего решающую ставку. Если сделать эту связь двунаправленной, вы сможете отобразить окно со списком всех аукционов, в которых победил конкретный пользователь: вместо запроса вы сможете вызвать метод `someUser.getBoughtItems()`.

С точки зрения класса `User` связь имеет тип *один ко многим*. Классы и их отношения показаны на рис. 8.8.

Рис. 8.8 ❖ Отношение «купил» между классами `User` и `Item`

Чем данная связь отличается от той, что связывает классы `Item` и `Bid`? Множественность *0..** в UML означает, что связь может быть необязательной. Это не повлияет на предметную модель, но будет иметь последствия для соответствующих таблиц. Можно ожидать наличия столбца внешнего ключа `BUYER_ID` в таблице `ITEM`. Столбец может содержать null, поскольку пользователь мог еще не купить конкретный товар, представленный объектом `Item` (пока аукцион еще проводится).

Можно принять тот факт, что столбец внешнего ключа может содержать `NULL`, и сформулировать дополнительные ограничения: «Может содержать `NULL`, только если аукцион еще не закончился или не было сделано ни одной ставки». В схемах реляционных баз данных мы всегда стараемся избегать столбцов, которые могут хранить null. Неопределенная информация снижает качество хранимой информации. Кортежи представляют истинные высказывания, и вы не можете утверждать того, чего не знаете. Более того, на практике многие разработчики и администраторы баз данных (DBA) не создают нужных ограничений, полагая, что целостность данных будет обеспечена несовершенным кодом приложения.

Необязательную связь между сущностями, будь то *один к одному* или *один ко многим*, лучше всего представлять в базе данных SQL таблицей соединения. Схема базы данных показана на рис. 8.9.

Рис. 8.9 ❖ Товары и покупатели связаны промежуточной таблицей

Ранее в этой главе мы уже добавляли таблицу соединения для организации связи *один к одному*. Чтобы гарантировать множественность *один к одному*, мы добавили ограничение уникальности для столбца внешнего ключа таблицы соединения. В данном случае мы имеем множественность *один ко многим*, поэтому только первичный ключ `ITEM_ID` должен быть уникальным: товар (`Item`) может быть куплен лишь однажды только одним пользователем (`User`). Значения столбца `BUYER_ID` не должны быть уникальными, поскольку один пользователь (`User`) может купить несколько товаров (`Item`).

Отображение коллекции `User#boughtItems` выглядит довольно просто:

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/jointable/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @OneToMany(mappedBy = "buyer")
    protected Set<Item> boughtItems = new HashSet<Item>();

    // ...
}
```

Это обычная сторона двунаправленной ассоциации, доступная только для чтения, действительное отображение которой на схему определено на отображающей стороне, перед свойством `Item#buyer`:

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/jointable/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinTable(
        name = "ITEM_BUYER",
        joinColumns =
            @JoinColumn(name = "ITEM_ID"),    <-- По умолчанию ID
        inverseJoinColumns =
            @JoinColumn(nullable = false)     <-- По умолчанию BUYER_ID
    )
    protected User buyer;

    // ...
}
```

Это ясно выраженное, необязательное отношение *один ко многим/многие к одному*. Если товар (`Item`) никто не купил, в таблице соединения `ITEM_BUYER` не будет соответствующей записи. В схеме нет проблемных столбцов, которые могли бы содержать null. Тем не менее для таблицы `ITEM_BUYER` нужно создать процедурное ограничение и триггер, выполняемый во время операции `INSERT`: «Разрешить

вставку строки с покупателем, если время аукциона для данного товара закончилось или пользователь сделал решающее предложение».

Далее следует последний пример со связью *один ко многим*. До сих пор нам встречались только связи *один ко многим*, направленные от сущности к сущности. Класс встраиваемого компонента также может быть связан с сущностью связью *один ко многим*.

8.2.4. Связь *один ко многим* во встраиваемых классах

Давайте снова рассмотрим отображение встраиваемого компонента, с которым мы работали на протяжении нескольких глав, – адреса (`Address`) пользователя (`User`). Мы расширим этот пример, добавив связь *один ко многим* от класса `Address` к классу `Shipment` – коллекцию с названием `deliveries` (доставки). UML-диаграмма классов для этой модели показана на рис. 8.10.

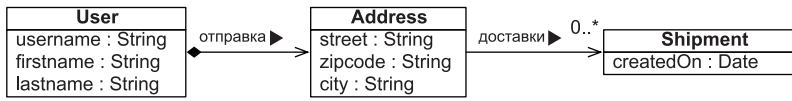


Рис. 8.10 ❖ Отношение *один ко многим* от класса `Address` к классу `Shipment`

Класс `Address` – это встраиваемый (`@Embeddable`) класс, а не сущность. Он может владеть однонаправленной связью к сущности; в данном случае связью с множественностью *один ко многим*, направленной к классу `Shipment` (в следующем разделе вы увидите встраиваемый класс, связанный с сущностью отношением *многие к одному*).

В классе `Address` объявлено свойство `Set<Shipment>`, представляющее эту ассоциацию:

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/embeddable/Address.java

```
@Embeddable
public class Address {

    @NotNull
    @Column(nullable = false)
    protected String street;

    @NotNull
    @Column(nullable = false, length = 5)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String city;

    @OneToMany
    @JoinColumn(
```

```
        name = "DELIVERY_ADDRESS_USER_ID",  ← По умолчанию DELIVERIES_ID
        nullable = false
    )
    protected Set<Shipment> deliveries = new HashSet<Shipment>();
    // ...
}
```

Первая стратегия отображения этой связи основана на аннотации `@JoinColumn`, определяющей столбец с именем `DELIVERY_ADDRESS_USER_ID`. Как видно на рис. 8.11, этот столбец с ограничением внешнего ключа расположен в таблице `SHIPMENT`.

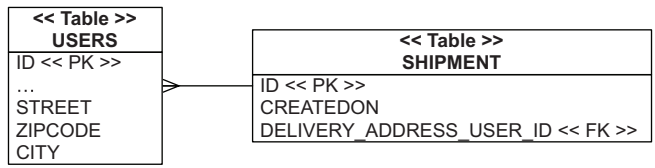


Рис. 8.11 ❖ Первичный ключ таблицы `USERS` связывает таблицы `USERS` и `SHIPMENT`

Встраиваемые компоненты не имеют собственных идентификаторов, поэтому значением столбца внешнего ключа будет идентификатор объекта `User`, в который встроен объект `Address`. Здесь мы также применили параметр `nullable = false`, чтобы каждой отправке `(Shipment)` соответствовал адрес доставки. Двунаправленная навигация в данном случае невозможна: экземпляр `Shipment` не может ссылаться на экземпляр `Address`, потому что встроенные компоненты не могут иметь общих ссылок.

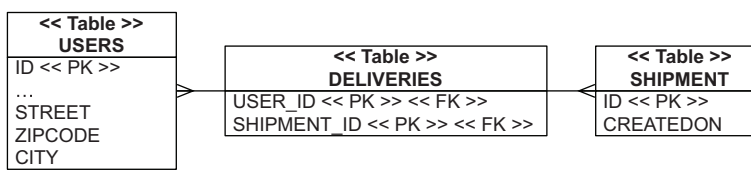


Рис. 8.12 ❖ Применение промежуточной таблицы для представления необязательной связи между таблицами `USERS` и `SHIPMENT`

Если связь необязательная и вы не желаете использовать столбец, который может содержать null, отобразите связь на промежуточную таблицу связи/соединения, как показано на рис. 8.12. Отображение коллекции в классе `Address` в этом случае будет определяться анноацией `@JoinTable` вместо `@JoinColumn`:

Файл: /model/src/main/java/org/jpwh/model/associations/onetomany/embeddablejointable/Address.java

```
@Embeddable
public class Address {

    @NotNull
```

```
    @Column(nullable = false)
    protected String street;

    @NotNull
    @Column(nullable = false, length = 5)
    protected String zipcode;

    @NotNull
    @Column(nullable = false)
    protected String city;

    @OneToMany
    @JoinTable(
        name = "DELIVERIES",   ← По умолчанию USERS_SHIPMENT
        joinColumns =
            @JoinColumn(name = "USER_ID"),      ← По умолчанию USERS_ID
        inverseJoinColumns =
            @JoinColumn(name = "SHIPMENT_ID")   ← По умолчанию SHIPMENTS_ID
    )
    protected Set<Shipment> deliveries = new HashSet<Shipment>();
    // ...
}
```

Обратите внимание: если не использовать аннотацию `@JoinTable` или `@JoinColumn`, связь `@OneToMany` во встраиваемом классе по умолчанию будет использовать стратегию с применением таблицы соединения.

Из класса сущности-владельца вы можете переопределять отображения свойств встроенного класса, используя аннотацию `@AttributeOverride`, как было показано в разделе 5.2.3. Чтобы переопределить отображение таблицы соединения или столбца, используйте аннотацию `@AssociationOverride` в классе сущности-владельца. Но вы не сможете изменить стратегию отображения; применение таблицы соединения или столбца определяется отображением в классе встраиваемого компонента.

Отображение таблицы соединения также подходит для полноценных отображений вида *многие ко многим*.

8.3. Тройные связи и связи *многие ко многим*

Связь между классами `Category` и `Item` имеет вид *многие ко многим*, как показано на рис. 8.13. В действующей системе у вас может не быть связи *многие ко многим*. Судя по нашему опыту, всегда есть дополнительная информация, сопутствующая каждой ссылке (link) между связанными экземплярами. В качестве примера можно привести время, когда товар (`Item`) был добавлен в категорию (`Category`), или пользователя (`User`), создавшего эту связь. Далее в этом разделе мы расширим наш пример, чтобы продемонстрировать такой случай. Начнем с обычной и простой связи *многие ко многим*.

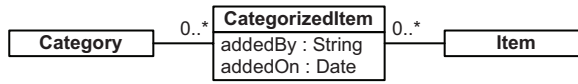


Рис. 8.13 ❖ Связь *многие ко многим* между классами `Category` и `Item`

8.3.1. Однонаправленные и двунаправленные связи *многие ко многим*

Таблицу соединения в базе данных, представляющую обычную связь *многие ко многим*, большинство разработчиков называет *связывающей таблицей*, или *таблицей связи*. На рис. 8.14 показано отношение *многие ко многим* со связывающей таблицей.

В связывающей таблице `CATEGORY_ITEM` есть два столбца, одновременно являющихся внешними ключами, ссылающимися на таблицы `CATEGORY` и `ITEM` соответственно. Первичный ключ таблицы включает оба столбца. Конкретная категория (`Category`) может быть связана с товаром (`Item`) только один раз, но каждый товар можно связать с несколькими категориями.

Используя JPA, можно отобразить связь *многие ко многим*, отметив коллекцию аннотацией `@ManyToMany`:

Файл: /model/src/main/java/org/jpwh/model/associations/manytomany/bidirectional/Category.java

```
@Entity
public class Category {
    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID")
    )
    protected Set<Item> items = new HashSet<Item>();

    // ...
}
```

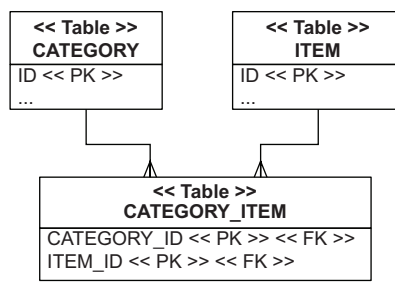


Рис. 8.14 ❖ Отношение *многие ко многим* со связывающей таблицей

Как обычно, чтобы упростить сохранение данных, используется параметр `CascadeType.PERSIST`. Когда приложение добавит в коллекцию ссылку на новый экземпляр `Item`, Hibernate сохранит его в базе данных. Давайте сделаем эту связь двунаправленной (этого можно не делать, если это не нужно):

Файл: /model/src/main/java/org/jpwh/model/associations/manytomany/bidirectional/Item.java

```
@Entity
public class Item {

    @ManyToMany(mappedBy = "items")
    protected Set<Category> categories = new HashSet<Category>();

    // ...
}
```

Как в любом отображении двунаправленной связи, одна сторона отображается другой стороной. Коллекция `Item#categories` по факту доступна только для чтения; во время сохранения данных Hibernate будет анализировать лишь содержимое коллекции `Category#items`. Далее создадим две категории и два товара, а затем свяжем их, используя отношение *многие ко многим*:

Файл: /examples/src/test/java/org/jpwh/test/associations/ManyToManyBidirectional.java

```
Category someCategory = new Category("Some Category");
Category otherCategory = new Category("Other Category");

Item someItem = new Item("Some Item");
Item otherItem = new Item("Other Item");

someCategory.getItems().add(someItem);
someItem.getCategories().add(someCategory);

someCategory.getItems().add(otherItem);
otherItem.getCategories().add(someCategory);

otherCategory.getItems().add(someItem);
someItem.getCategories().add(otherCategory);

em.persist(someCategory);
em.persist(otherCategory);
```

Благодаря использованию транзитивного сохранения запись двух категорий сохранит весь граф экземпляров. С другой стороны, параметры каскадирования `ALL`, `REMOVE`, а также удаление осиротевших объектов (см. раздел 7.3.3) в случае связи *многие ко многим* бессмысленны. Вот и наступил подходящий момент, чтобы проверить, разбираетесь ли вы в сущностях и типах-значениях. Попробуйте самостоятельно найти разумные объяснения, почему эти типы каскадирования не имеют смысла для связи *многие ко многим*.

Можно ли вместо множества (`Set`) использовать список (`List`) или даже контейнер? Множество (`Set`) прекрасно согласуется со схемой базы данных, поскольку там тоже нет дублирующих ссылок между классами `Category` и `Item`.

Контейнер позволяет дублировать элементы, следовательно, в таблице соединения понадобится определить другой первичный ключ. Этого можно добиться при помощи аннотации `@CollectionId`, показанной в разделе 7.1.5. Одна из альтернативных стратегий отображения связи *многие ко многим*, которую мы обсудим через мгновение, является лучшим решением для поддержки дублирующих ссылок.

Отображать упорядоченные коллекции, такие как `List`, можно с помощью обычной аннотации `@ManyToMany`, но лишь с одной стороны. Помните, что в двунаправленном отношении одна сторона отображается другой стороной, т. е. Hibernate будет игнорировать ее значение при синхронизации с базой данных. Если обе стороны представлены списками, сделать индексы элементов хранимыми можно только с одной стороны отношения.

Обычная аннотация `@ManyToMany` скрывает связывающую таблицу; она не имеет соответствующего Java-класса; есть только свойства-коллекции. Поэтому когда кто-то скажет: «Мне нужно больше столбцов с информацией о связи в моей связывающей таблице», – а судя по нашему опыту, рано или поздно вы обязательно услышите такие слова, – отобразите эту информацию в классе Java.

8.3.2. Связь *многие ко многим* с промежуточной сущностью

Связь *многие ко многим* всегда можно представить в виде двух связей *многие к одному* с промежуточным классом. В этом случае промежуточная таблица не скрывается, а явно определяется с помощью класса Java. Такую модель, как правило, легче расширять, поэтому мы стараемся не использовать в приложениях обычную связь *многие ко многим*. Позднее, когда в связывающую таблицу добавятся дополнительные столбцы, будет гораздо труднее провести рефакторинг; поэтому, прежде чем применять отображение `@ManyToMany`, как показано в предыдущем разделе, рассмотрите альтернативное решение, показанное на рис. 8.15.

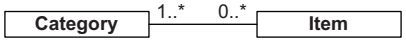


Рис. 8.15 ❖ Класс `CategorizedItem`, соединяющий классы `Category` и `Item`

Представьте, что вам нужно сохранять информацию при каждом связывании товара (`Item`) с категорией (`Category`). Класс `CategorizedItem` описывает время и пользователя, создавшего связь. Эта модель требует наличия дополнительных столбцов в таблице соединения, как видно на рис. 8.16.

Новая сущность `CategorizedItem` будет отображаться на связывающую таблицу, как показано далее.

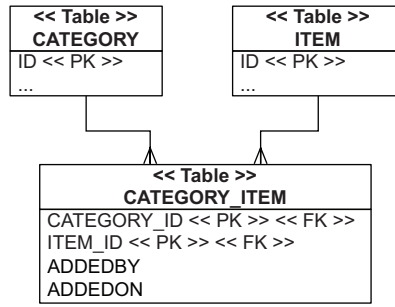


Рис. 8.16 ❖ Дополнительные столбцы в таблице соединения для отношения *многие ко многим*

Листинг 8.4 ❖ Отображение отношения многие ко многим с помощью класса `CategorizedItem`

Файл: /model/src/main/java/org/jpwh/model/associations/manytomany/linkentity/ CategorizedItem.java

```
@Entity
@Table(name = "CATEGORY_ITEM")
@org.hibernate.annotations.Immutable  ← ❶ Объявляет класс неизменяемым
public class CategorizedItem {

    @Embeddable
    public static class Id implements Serializable {  ← ❷ Инкапсулирует составной ключ

        @Column(name = "CATEGORY_ID")
        protected Long categoryId;

        @Column(name = "ITEM_ID")
        protected Long itemId;

        public Id() {
        }

        public Id(Long categoryId, Long itemId) {
            this.categoryId = categoryId;
            this.itemId = itemId;
        }

        public boolean equals(Object o) {
            if (o != null && o instanceof Id) {
                Id that = (Id) o;
                return this.categoryId.equals(that.categoryId)
                    && this.itemId.equals(that.itemId);
            }
            return false;
        }

        public int hashCode() {
            return categoryId.hashCode() + itemId.hashCode();
```

```
        }
    }

    @EmbeddedId  ← ❸ Отображает свойство идентификатора и столбцы составного ключа
    protected Id id = new Id();

    @Column(updatable = false)
    @NotNull
    protected String addedBy;  ← ❹ Отображает имя пользователя

    @Column(updatable = false)
    @NotNull
    protected Date addedOn = new Date();  ← ❺ Отображает время

    @ManyToOne
    @JoinColumn(
        name = "CATEGORY_ID",
        insertable = false, updatable = false)
    protected Category category;  ← ❻ Отображает категорию

    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        insertable = false, updatable = false)
    protected Item item;  ← ❼ Отображает товар

    public CategorizedItem(
        String addedByUsername,  ← ❽ Конструирует объект CategorizedItem
        Category category,
        Item item) {

        this.addedBy = addedByUsername;   | Присваивание
        this.category = category;         | значений
        this.item = item;                 | полям класса

        this.id.categoryId = category.getId();   | Установка значения
        this.id.itemId = item.getId();           | идентификатора

        category.getCategorizedItems().add(this);   ← Гарантирует ссылочную целостность, если отношение двунаправленное
        item.getCategorizedItems().add(this);
    }

    // ...
}
```

Это довольно большой фрагмент кода с новыми аннотациями. Во-первых, это неизменяемый класс сущности, следовательно, вы не сможете обновить значения свойств объекта после создания. Если объявить класс неизменяемым, Hibernate сможет применить некоторые оптимизации, например не проверять состояния объектов во время выталкивания контекста хранения ❶.

Класс сущности должен иметь свойство идентификатора. Первичный ключ связывающей таблицы включает столбцы `CATEGORY_ID` и `ITEM_ID`. Поэтому

в классе сущности также объявлен составной ключ, который для удобства инкапсулирован во вложенном классе встраиваемого компонента ❷. При желании этот класс можно поместить в отдельный файл. Новая аннотация `@EmbeddedId` ❸ отображает свойство идентификатора и столбцы составного ключа в таблицу сущности.

Далее следуют два свойства, отображающие имя пользователя (`addedBy`) ❹ и время (`addedOn`) ❺ в столбцы таблицы соединения. Это и есть та самая «дополнительная информация о связи».

Два поля с аннотацией `@ManyToOne` – `category` ❻ (категория) и `item` ❼ (товар) – отображаются в столбцы, уже перечисленные в настройках отображения идентификатора. Интересно отметить, что здесь с помощью параметров `updatable=false` и `insertable=false` эти свойства объявляются доступными только для чтения. Благодаря этому Hibernate будет записывать значения столбцов, используя значение идентификатора экземпляра `CategorizedItem`. В то же время вы сможете читать и просматривать связанные экземпляры, вызывая `categorizedItem.getItem()` и `getCategory()` соответственно (если отобразить один столбец дважды, не ограничив доступа в одном из отображений, при запуске приложения Hibernate пожалуется на дублирование отображений столбца).

Также во время создания экземпляра `CategorizedItem` ❽ происходит присваивание значений идентификатора: приложение всегда должно устанавливать значения составного ключа; Hibernate не будет их генерировать. Обратите особое внимание на конструктор – как он устанавливает значения полей класса и обеспечивает ссылочную целостность, отображая коллекции по обеим сторонам ассоциации. Мы отобразим эти коллекции далее, чтобы обеспечить двунаправленную навигацию.

Этого однонаправленного отображения достаточно для поддержки отношения *многие ко многим* между классами `Category` и `Item`. Для создания связи нужно сконструировать и сохранить объект `CategorizedItem`. Чтобы разрушить связь, удалите экземпляр `CategorizedItem`. Конструктору класса `CategorizedItem` требуется предоставить уже хранящиеся экземпляры классов `Category` и `Item` соответственно.

Если требуется реализовать двунаправленную навигацию, отобразите коллекцию, используя аннотацию `@OneToMany` в классе `Category` и/или `Item`:

Файл: /model/src/main/java/org/jpwh/model/associations/manytomany/linkentity/ Category.java

```
@Entity
public class Category {

    @OneToMany(mappedBy = "category")
    protected Set<CategorizedItem> categorizedItems = new HashSet<>();

    // ...
}
```

Файл: /model/src/main/java/org/jpwh/model/associations/manytomany/linkentity/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    protected Set<CategorizedItem> categorizedItems = new HashSet<>();

    // ...
}
```

Каждая из сторон отображается аннотациями в классе `CategorizedItem`, поэтому Hibernate уже знает, что делать при выполнении итераций по элементам коллекции, возвращаемой любым из методов `getCategorizedItems()`.

Вот как создаются и сохраняются связи:

Файл: /examples/src/test/java/org/jpwh/test/associations/ManyToManyLinkEntity.java

```
Category someCategory = new Category("Some Category");
Category otherCategory = new Category("Other Category");
em.persist(someCategory);
em.persist(otherCategory);

Item someItem = new Item("Some Item");
Item otherItem = new Item("Other Item");
em.persist(someItem);
em.persist(otherItem);

CategorizedItem linkOne = new CategorizedItem(
    "johndoe", someCategory, someItem
);

CategorizedItem linkTwo = new CategorizedItem(
    "johndoe", someCategory, otherItem
);

CategorizedItem linkThree = new CategorizedItem(
    "johndoe", otherCategory, someItem
);

em.persist(linkOne);
em.persist(linkTwo);
em.persist(linkThree);
```

Основное преимущество этой стратегии состоит в возможности двунаправленной навигации: вы можете получить все товары, относящиеся к данной категории, вызвав `someCategory.getCategorizedItems()`, а затем осуществить навигацию в обратном направлении с помощью `someItem.getCategorizedItems()`. Недостатком является более сложный код управления экземплярами сущностей `CategorizedItem`, которые нужно создавать и удалять независимо. Также требуется определить

в классе `CategorizedItem` некоторую инфраструктуру, такую как составной идентификатор. Небольшим улучшением могло бы стать применение параметра `CascadeType.PERSIST` для некоторых ассоциаций, что снизило бы число вызовов метода `persist()`.

В предыдущем примере мы сохраняли пользователя, создавшего связь между экземплярами `Category` и `Item`, в виде обычной строки с именем. Если вместо этого создать в таблице соединения столбец внешнего ключа `USER_ID`, получится тройное отношение. Класс `CategorizedItem` будет находиться в отношении `@ManyToOne` с классами `Category`, `Item` и `User`.

В следующем разделе вы познакомитесь с еще одной стратегией отображения *многие ко многим*. Чтобы сделать пример более интересным, создадим тройную связь.

8.3.3. Тройные связи с компонентами

В предыдущем разделе мы представляли отношение *многие ко многим* с помощью класса сущности, отображаемого на связывающую таблицу. Возможно, что более простой альтернативой было бы отображение класса встраиваемого компонента:

Файл: /model/src/main/java/org/jpwh/model/associations/manytomany/ternary/CategorizedItem.java

```
@Embeddable
public class CategorizedItem {

    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        nullable = false, updatable = false
    )
    protected Item item;

    @ManyToOne
    @JoinColumn(
        name = "USER_ID",
        updatable = false
    )
    @NotNull
    protected User addedBy;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(updatable = false)
    @NotNull
    protected Date addedOn = new Date();

    protected CategorizedItem() {
    }

    public CategorizedItem(User addedBy,
                           Item item) {
```

Не формирует ограничения SQL и, следовательно, не входит в состав первичного ключа

```
        this.addedBy = addedBy;
        this.item = item;
    }

    // ...
}
```

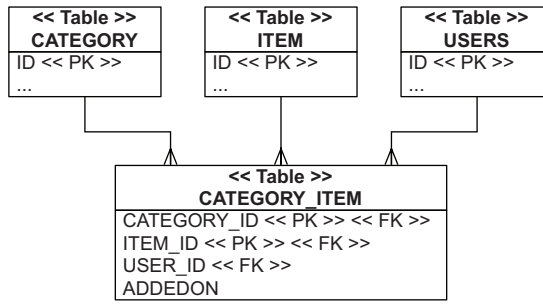


Рис. 8.17 ❖ Связывающая таблица с тремя столбцами внешних ключей

Здесь появилось новое отображение связи `@ManyToOne` во встраиваемом (`@Embeddable`) классе, а также дополнительный столбец соединения с внешним ключом `USER_ID`, создающий тройное отношение. Взгляните на схему базы данных на рис. 8.17.

Владельцем коллекции встраиваемых компонентов является сущность `Category`:

Файл: /model/src/main/java/org/jpwh/model/associations/manytomany/ternary/Category.java

```
@Entity
public class Category {

    @ElementCollection
    @CollectionTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID")
    )
    protected Set<CategorizedItem> categorizedItems = new HashSet<>();

    // ...
}
```

К сожалению, данное отображение несовершенно: при отображении коллекции (`@ElementCollection`) встраиваемого типа все свойства целевого типа, имеющие параметр `nullable=false`, становятся частью (составного) первичного ключа. Хочется, чтобы все столбцы таблицы `CATEGORY_ITEM` имели ограничение `NOT NULL`. Но в состав первичного ключа должны входить только столбцы `CATEGORY_ID` и `ITEM_ID`.

Для этого все свойства, не входящие в состав первичного ключа, следует отметить аннотацией `@NotNull` из Bean Validation. В таком случае (поскольку это встраиваемый класс) Hibernate проигнорирует аннотации Bean Validation для реализации первичного ключа и во время генерации SQL-схемы. Однако в этом случае столбцы `USER_ID` и `ADDEDON` в сформированной схеме не получат соответствующих ограничений `NOT NULL`, которые придется добавлять вручную.

Преимуществом этой стратегии является неявный жизненный цикл связанных компонентов. Для создания связи между экземплярами `Category` и `Item` нужно добавить в коллекцию новый экземпляр `CategorizedItem`. Для разрушения связи достаточно удалить элемент из коллекции. Дополнительных настроек каскадирования не требуется, а Java-код стал проще (хотя и занимает больше строк):

Файл: /examples/src/test/java/org/jpwh/test/associations/ManyToManyTernary.java

```
Category someCategory = new Category("Some Category");
Category otherCategory = new Category("Other Category");
em.persist(someCategory);
em.persist(otherCategory);

Item someItem = new Item("Some Item");
Item otherItem = new Item("Other Item");
em.persist(someItem);
em.persist(otherItem);

User someUser = new User("johndoe");
em.persist(someUser);

CategorizedItem linkOne = new CategorizedItem(
    someUser, someItem
);
someCategory.getCategorizedItems().add(linkOne);

CategorizedItem linkTwo = new CategorizedItem(
    someUser, otherItem
);
someCategory.getCategorizedItems().add(linkTwo);

CategorizedItem linkThree = new CategorizedItem(
    someUser, someItem
);
otherCategory.getCategorizedItems().add(linkThree);
```

Двунаправленная навигация в этой ситуации невозможна: такой встраиваемый компонент, как `CategorizedItem`, не поддерживает общих ссылок по определению. Также невозможно осуществить переход от объекта `Item` к объекту `CategorizedItem`, поэтому в классе `Item` нет отображения этой связи. Вместо этого можно написать запрос для извлечения всех категорий данного товара (`Item`):

Файл: /examples/src/test/java/org/jpwh/test/associations/ManyToManyTernary.java

```
Item item = em.find(Item.class, ITEM_ID);

List<Category> categoriesOfItem =
    em.createQuery(
        "select c from Category c " +
            "join c.categorizedItems ci " +
            "where ci.item = :itemParameter")
    .setParameter("itemParameter", item)
    .getResultList();

assertEquals(categoriesOfItem.size(), 2);
```

Вот мы и завершили ваше первое отображение тройной связи. В предыдущих главах вы видели примеры работы ORM со словарями; ключи и значения всегда имели простой или встраиваемый тип. В следующем разделе вы познакомитесь с более сложными типами пар ключ/значение и их отображениями.

8.4. Связи между сущностями с использованием словарей

Ключами и значениями словарей могут быть другие сущности, что дает другую стратегию отображения связи *многие ко многим* и тройного отношения. Для начала предположим, что ссылкой на сущность будет только значение в каждом элементе словаря.

8.4.1. Связь *один ко многим* со свойством для ключа

Если значением каждого элемента словаря будет ссылка на другую сущность, получится отношение *один ко многим* между сущностями. Ключ словаря имеет простой тип, например `Long`.

Примером такой структуры может служить сущность `Item` со словарем экземпляров `Bid`, где каждый элемент является парой из идентификатора объекта `Bid` и ссылки на него. Во время обхода коллекции `someItem.getBids()` фактически будут выполняться итерации по элементам словаря, имеющим вид `(1, <ссылка на объект Bid с первичным ключом 1>)`, `(2, <ссылка на объект Bid с первичным ключом 2>)` и т. д.:

Файл: /examples/src/test/java/org/jpwh/test/associations/MapsMapKey.java

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(item.getBids().size(), 2);

for (Map.Entry<Long, Bid> entry : item.getBids().entrySet()) {
    assertEquals(entry.getKey(),
     entry.getValue().getId());  ← В роли ключа выступает идентификатор объекта Bid
}
```

Таблицы, реализующие данное отображение, не представляют ничего особенного; у нас есть таблицы ITEM и BID, со столбцом внешнего ключа ITEM_ID в таблице BID. Это та же схема, что была показана на рис. 7.14 для отображения связи *один ко многим/многие к одному*, использующая обычную коллекцию вместо словаря (Map). Нашей целью является несколько иное представление данных в приложении.

Добавим в класс Item поле bids типа Map:

Файл: /model/src/main/java/org/jpwh/model/associations/maps/mapkey/Item.java

```
@Entity
public class Item {

    @MapKey(name = "id")
    @OneToMany(mappedBy = "item")
    protected Map<Long, Bid> bids = new HashMap<>();

    // ...
}
```

Здесь появилась новая аннотация @MapKey. Она отображает поле целевой сущности – в данном случае Bid – как ключ словаря. Если не использовать атрибута name, по умолчанию будет выбрано имя свойства-идентификатора целевой сущности, поэтому в данном примере параметр name не нужен. Поскольку ключи словаря образуют множество, можно ожидать, что значения конкретного словаря будут уникальными. Это справедливо для первичных ключей экземпляров Bid, но вряд ли для каких-нибудь других полей класса Bid. Только вы можете гарантировать уникальность значений выбранного свойства – Hibernate не будет делать проверку.

В основном, и довольно редко, данный способ отображения применяется, когда требуется реализовать обход элементов словаря, используя в качестве ключа какое-либо свойство сущности, которая является значением записи, поскольку так, возможно, удобнее отображать данные. Чаще словари применяются в середине тройной ассоциации.

8.4.2. Тройное отношение вида ключ/значение

Вам, должно быть, уже скучно, но мы обещаем, что это будет последний раз, когда мы покажем еще один способ отображения связи между классами Category и Item. Ранее в разделе 8.3.3 для представления связи мы использовали встраиваемый компонент CategorizedItem. Здесь мы покажем, как представить отношение с помощью словаря вместо дополнительного класса Java. Ключом каждого элемента будет служить товар (Item), а связанным значением – пользователь (User), добавивший товар (Item) в категорию (Category), как показано на рис. 8.18.

Таблица соединения/связи, как показано на рис. 8.19, имеет три столбца: CATEGORY_ID, ITEM_ID и USER_ID. Коллекцией Map владеет сущность Category:

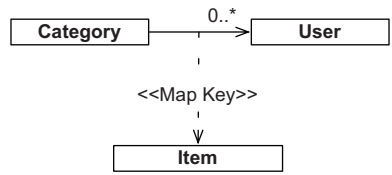


Рис. 8.18 ❖ Коллекция `Map` со связями в виде пар ключ/значение

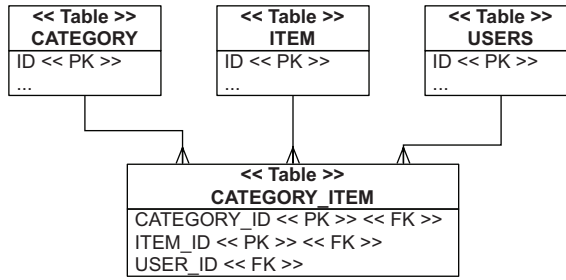


Рис. 8.19 ❖ Связывающая таблица представляет пары ключ/значение коллекции `Map`

Файл: /model/src/main/java/org/jpwh/model/associations/maps/ternary/Category.java

```
@Entity
public class Category {

    @ManyToMany(cascade = CascadeType.PERSIST)
    @MapKeyJoinColumn(name = "ITEM_ID")   ← По умолчанию ITEMADDEDBY_KEY
    @JoinTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID"),
        inverseJoinColumns = @JoinColumn(name = "USER_ID")
    )
    protected Map<Item, User> itemAddedBy = new HashMap<>();

    // ...
}
```

Аннотацию `@MapKeyJoinColumn` можно опустить; по умолчанию Hibernate сгенерирует имя `ITEMADDEDBY_KEY` для столбца соединения/внешнего ключа, ссылающегося на таблицу `ITEM`.

К моменту создания связи между всеми тремя сущностями каждая из них должна храниться в базе данных:

Файл: /examples/src/test/java/org/jpwh/test/associations/MapsTernary.java

```
someCategory.getItemAddedBy().put(someItem, someUser);
someCategory.getItemAddedBy().put(otherItem, someUser);
otherCategory.getItemAddedBy().put(someItem, someUser);
```

Для разрушения связи удалите элемент из словаря. Это удобный Java API для управления сложными отношениями, скрывающий связывающую таблицу с тремя столбцами в базе данных. Но помните, что на практике связывающие таблицы часто обрастают дополнительными столбцами, и последующий рефакторинг кода Java-приложения может оказаться затратным, если вы зависите от API коллекции `Map`. Ранее у нас был столбец `ADDEDON` со значением времени создания связи, но для этого отображения нам пришлось его убрать.

8.5. Резюме

- Вы узнали, как отображать сложные отношения между сущностями, используя связи *один к одному*, *один ко многим*, *многие ко многим*, тройные связи и связи сущностей со словарями.
- Упрощайте отношения между классами, и вам редко понадобится использовать большую часть показанных способов отображения. В частности, всегда лучше представить связь *многие ко многим* в виде двух связей *многие к одному* из промежуточного класса сущности либо как коллекцию компонентов.
- Прежде чем браться за сложное отображение, убедитесь, что коллекция действительно необходима. Спросите себя, как часто потребуется выполнять итерации по всем элементам.
- Структуры Java, показанные в этой главе, порой могут упростить доступ к данным, но, как правило, усложняют хранение, изменение и удаление.

Глава 9

Сложные и унаследованные схемы

В этой главе:
- улучшение схемы SQL с помощью дополнительных инструкций DDL;
- взаимодействие с унаследованной базой данных;
- отображение составных ключей.

В этой главе мы рассмотрим самую важную часть вашей системы – схему базы данных, которая содержит набор правил целостности, – созданную вами модель реального мира. Если в реальных условиях в вашем приложении можно выставлять товар на торги только один раз, схема базы данных должна гарантировать это условие. Если на аукционе всегда есть начальная цена, модель вашей базы данных должна содержать соответствующее ограничение. Если данные соответствуют всем правилам целостности, они называются *согласованными* – термин, который вы еще встретите в разделе 11.1.

Мы также предполагаем, что согласованные данные *корректны*: все утверждения в базе данных, явные или неявные, являются истинными; все остальное ложно. Если вам хочется узнать больше о теории, стоящей за этим подходом, прочтите о *предположении о замкнутости мира*[1].

Главные нововведения в JPA 2

- Формирование схемы и выполнение произвольных сценариев SQL во время загрузки теперь стандартизировано и может быть настроено в рамках единицы хранения.
- Можно отображать и настраивать элементы схемы, такие как имена индексов и внешних ключей, используя стандартные аннотации.

[1] http://wiki-org.ru/wiki/Предположение_о_замкнутости_мира. – *Прим. ред.*

- Внешние ключи и связи *многие к одному* можно отображать с помощью составного первичного ключа в виде «производной сущности», используя аннотацию `@MapsId`.

Иногда есть возможность начать проектирование сверху вниз. Нет ни схемы базы данных, ни, скорее всего, самих данных – ваше приложение совсем новое. Многие разработчики с удовольствием позволяют Hibernate автоматически сгенерировать сценарии для создания схемы базы данных. Также вы, скорее всего, позволите Hibernate развернуть схему в тестовой базе данных на вашем компьютере для разработки или в вашей системе непрерывного развертывания для выполнения интеграционных тестов. Позже администратор базы данных возьмет сгенерированные сценарии и создаст окончательную и улучшенную схему для развертывания программы в рабочей среде. В первой части этой главы вы увидите, как улучшить схему, используя JPA и Hibernate, чтобы сделать вашего администратора базы данных счастливым.

На другом конце спектра находятся системы с уже существующими и, вероятно, сложными схемами и данными, которые копились годами. Ваше новое приложение станет лишь маленьким винтиком в огромном механизме, и ваш администратор базы данных не позволит внести никаких (даже самых безобидных) изменений в базу данных. Вам понадобится гибкое объектно-реляционное отображение, чтобы не пришлось потом вносить слишком много изменений в классы Java, если не все будет получаться сразу. Это станет темой второй половины данной главы наряду с обсуждением составных первичных и внешних ключей.

Мы начнем с совершенно новой реализации и рассмотрим схемы, сгенерированные Hibernate.

9.1. Улучшаем схему базы данных

Hibernate читает классы предметной модели и метаданные отображения и генерирует инструкции DDL, формирующие схему базы данных. Их можно экспортировать в текстовый файл или выполнять непосредственно в базе данных каждый раз, когда проводится интеграционное тестирование. Поскольку язык описания схемы в значительной степени зависит от используемой базы данных, каждый параметр в метаданных отображения может привязать вас к определенной системе; помните об этом при определении схемы.

Hibernate автоматически создает основную схему для таблиц и ограничений; он даже определяет последовательности в зависимости от выбранного генератора идентификаторов. Но некоторые элементы схемы Hibernate не может и не будет создавать автоматически. К ним относятся настройки производительности, характерные для конкретной базы данных, и прочие параметры, относящиеся к физическому хранению данных (например, пространства таблиц). Помимо физических аспектов, ваш администратор базы данных обычно предоставляет дополнительные инструкции для улучшения схемы. Администраторы баз данных должны

включаться в процесс разработки как можно раньше для проверки схем, автоматически сгенерированных Hibernate. Никогда не запускайте приложение в рабочей среде, не проверив автоматически сгенерированную схему.

Изменения, сделанные администратором базы данных, могут передаваться обратно в систему Java для добавления в метаданные отображения, если это позволяет процесс разработки. Во многих проектах метаданные могут содержать все необходимые изменения, полученные от администратора, и на их основе Hibernate сможет сгенерировать окончательную рабочую схему в рамках обычной сборки, включая все комментарии, ограничения, индексы и т. д.

В следующих разделах мы покажем, как настроить сгенерированную схему и как добавить вспомогательные артефакты схемы базы данных (иногда мы называем их *объектами*; здесь мы не имеем в виду объектов Java). Мы обсудим пользовательские типы данных, дополнительные правила целостности, индексы и то, как изменить имена некоторых артефактов (иногда довольно страшные), автоматически сгенерированные Hibernate.

Экспорт сценария создания схемы в файл

Hibernate связывает класс `org.hibernate.tool.hbm2ddl.SchemaExport` с методом `main()`, который можно вызвать из командной строки. Эта утилита может обращаться к вашей СУБД напрямую, чтобы создать схему, или записывать текстовый файл со сценарием DDL для дальнейшей доработки администратором базы данных.

Сначала рассмотрим, как внедрить произвольные SQL-выражения в процесс автоматической генерации схемы фреймворком Hibernate.

9.1.1. Добавление вспомогательных объектов базы данных

В процесс генерации схемы можно внедрить три типа SQL-сценариев:

- сценарий *создания* выполняется во время генерации схемы. Такой сценарий может быть выполнен до, после или вместо сценария, автоматически сгенерированного фреймворком Hibernate. Другими словами, можно написать сценарий SQL, который будет выполняться до или после того, как Hibernate сгенерирует таблицы, ограничения и пр. из метаданных отображения;
- сценарий *удаления* выполняется, когда Hibernate удаляет элементы схемы. Подобно сценарию создания, сценарий удаления может выполняться до, после или вместо инструкций, автоматически сгенерированных Hibernate;
- сценарий *загрузки* всегда выполняется на заключительном этапе создания, после того как Hibernate сгенерирует схему. Его главная цель – импортировать тестовые или основные данные перед запуском приложения или модульных тестов. Он может содержать любые типы инструкций SQL, включая такие DDL-выражения, как `ALTER`, если требуется дальнейшая модификация схемы.

Подобная модификация процесса генерации схемы на самом деле стандартизована; она настраивается спомощью параметров JPA в файле настроек единицы хранения persistence.xml.

Листинг 9.1 ❖ Настройки генерации схемы в файле persistence.xml

Файл: /model/src/main/resources/META-INF/persistence.xml

```
<property name="hibernate.0hbm2ddl.import_files_sql_extractor"
          value="org.hibernate.tool.hbm2ddl.
          ➥MultipleLinesSqlCommandExtractor"/>

<property name="javax.persistence.schema-generation.create-source"
          value="script-then-metadata"/>
<property name="javax.persistence.schema-generation.drop-source"
          value="metadata-then-script"/>

<property name="javax.persistence.schema-generation.create-script-source"
          value="complexschemas/CreateScript.sql.txt"/>

<property name="javax.persistence.schema-generation.drop-script-source"
          value="complexschemas/DropScript.sql.txt"/>

<property name="javax.persistence.sql-load-script-source"
          value="complexschemas/LoadScript.sql.txt"/>
```

❶ Активирует механизм многострочного извлечения

❷ Определяет, когда должны выполняться сценарии

❸ Произвольный сценарий SQL для создания схемы

❹ Произвольный сценарий SQL для удаления схемы

❺ Сценарий загрузки

❶ По умолчанию Hibernate предполагает, что каждая строка сценария содержит отдельное выражение SQL. Эта настройка позволяет воспользоваться более удобным многострочным экстрактором. В сценариях выражения SQL завершаются точкой с запятой. Вы можете создать собственную реализацию `org.hibernate.tool.hbm2ddl.ImportSqlCommandExtractor`, если хотите обрабатывать сценарии SQL иным способом.

❷ Эта настройка определяет, когда выполняются сценарии создания и удаления. Ваши сценарии SQL будут содержать инструкции создания доменов `CREATE DOMAIN`, которые должны выполниться перед созданием таблиц, использующих эти домены. С такими настройками генератор схемы выполнит сценарий создания, прежде чем начнет чтение метаданных ORM (аннотаций, файлов XML) и создание таблиц. Сценарий удаления выполнится после того, как Hibernate удалит таблицы, предоставляя вам возможность удалить все, что было создано вами. Другими вариантами являются `metadata` (игнорировать ваши сценарии) и `script` (использовать только ваши сценарии и игнорировать метаданные ORM в аннотациях и файлах XML).

❸ Местоположение сценария SQL для создания схемы. Путь – это (а) местоположение сценария в пути поиска классов; (б) местоположение сценария в виде указателя ресурса URL со схемой file:// и (в) абсолютный или относительный путь в локальной файловой системе. В данном примере используется вариант «а».

❹ Сценарий SQL для удаления схемы.

❺ Сценарий загрузки, который выполнится после создания всех таблиц.

Мы уже говорили, что инструкции DDL обычно сильно зависят от конкретной базы данных. Если приложение должно работать с несколькими разновидностями баз данных, вам может понадобиться несколько наборов со сценариями создания, удаления и загрузки для настройки схемы под каждую конкретную разновидность

базы данных. Эту проблему можно решить, используя несколько определений единиц хранения в файле persistence.xml.

Также можно использовать собственные настройки Hibernate для модификации схемы в файле отображения hbm.xml.

Листинг 9.2 ❖ Генерация схемы с применением нестандартной конфигурации Hibernate

```
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <database-object>
        <create>
            CREATE ...
        </create>
        <drop>
            DROP ...
        </drop>
        <dialect-scope name="org.hibernate.dialect.H2Dialect"/>
        <dialect-scope name="org.hibernate.dialect.PostgreSQL82Dialect"/>
    </database-object>
</hibernate-mapping>
```

Особенности Hibernate

Поместите свои инструкции SQL внутрь элементов `<create>` и `<drop>`. А Hibernate выполнит их *после* создания схемы для классов предметной модели, т. е. после создания таблиц и *до* удаления автоматически сгенерированной части схемы. Это поведение фиксировано, поэтому стандартные настройки генерации схемы JPA часто оказываются более гибкими.

Элемент `<dialect-scope>` ограничивает инструкции SQL определенным набором настроенных диалектов баз данных. Когда элемент `<dialect-scope>` отсутствует, выражения SQL выполняются всегда.

Фреймворк Hibernate также поддерживает сценарии загрузки: обнаружив файл с именем import.sql в корне каталога классов, он выполнит его по завершении создания схемы. Если у вас есть несколько файлов для импорта, укажите их имена в виде списка, разделенного запятыми, в параметре `hibernate.hbm2ddl.import_files`, в конфигурации единицы хранения.

Наконец, если потребуется программно управлять сгенерированной схемой, реализуйте интерфейс `org.hibernate.mapping.AuxiliaryDatabaseObject`. В Hibernate уже есть готовая реализация для удобства наследования, поэтому можно создать свой подкласс, выборочно переопределив необходимые методы.

Листинг 9.3 ❖ Программное управление сгенерированной схемой

```
package org.jpwh.model.complexschemas;

import org.hibernate.dialect.Dialect;
import org.hibernate.boot.model.relational.AbstractAuxiliaryDatabaseObject;

public class CustomSchema
```

```
    extends AbstractAuxiliaryDatabaseObject {

    public CustomSchema() {
        addDialectScope("org.hibernate.dialect.Oracle9Dialect");
    }

    @Override
    public String[] sqlCreateStrings(Dialect dialect) {
        return new String[]{"[CREATE statement]"};
    }

    @Override
    public String[] sqlDropStrings(Dialect dialect) {
        return new String[]{"[DROP statement]"};
    }
}
```

С помощью методов `sqlCreateString()` и `sqlDropString()` можно программно добавлять используемые диалекты и даже получать доступ к некоторым данным отображения. Этот класс следует объявить в файле hbm.xml:

```
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <database-object>
        <definition class="org.jpwh.model.complexschemas.CustomSchema"/>
        <dialect-scope name="org.hibernate.dialect.H2Dialect"/>
        <dialect-scope name="org.hibernate.dialect.PostgreSQL82Dialect"/>
    </database-object>

</hibernate-mapping>
```

Дополнительные диалекты суммируются; предыдущий пример добавляет три диалекта.

Давайте напишем собственные сценарии создания, удаления и загрузки и реализуем дополнительные правила целостности, которые порекомендовал бы каждый хороший администратор баз данных. Но сначала немного справочной информации о правилах целостности и ограничениях SQL.

9.1.2. Ограничения SQL

Системы, проверяющие целостность данных лишь в коде приложения, могут приводить к повреждению данных и со временем снижают качество работы базы данных. Если хранилище данных не гарантирует выполнения правил, простейшая незамеченная ошибка в приложении сможет вызвать такие проблемы, которые нельзя будет исправить, например потерю или повреждение данных.

Вместо проверки целостности данных в коде процедурного (или объектно-ориентированного) приложения СУБД позволяют декларативно реализовать правила целостности в схеме базы данных. Преимущества декларативных правил заключаются в потенциальном уменьшении ошибок в коде и возможности для СУБД оптимизировать доступ к данным.

В базах данных SQL мы различаем четыре типа правил:

- *ограничения домена* (domain constraints). Домен – это (грубо говоря, и только в области баз данных) тип данных в базе. Следовательно, ограничение домена определяет диапазон возможных значений конкретного типа данных. Например, тип данных `INTEGER` подходит для целочисленных значений. Тип данных `CHAR` может хранить строки символов: к примеру, всех символов, определенных в кодировке ASCII или какой-то другой. Используя в основном встроенные типы данных СУБД, мы полагаемся на ограничения домена, определенные разработчиком СУБД. Если ваша база данных обладает необходимой функциональностью, вы могли бы воспользоваться (очень ограниченной) поддержкой пользовательских доменов, чтобы добавить дополнительные ограничения к существующим типам данных или создать собственные типы данных;
- *ограничения столбца* (column constraints). Ограничение области допустимых значений столбца определенным доменом и типом формирует ограничение столбца. Например, в схеме можно объявить столбец `EMAIL`, содержащий значения типа `VARCHAR`. С другой стороны, можно создать новый домен `EMAIL_ADDRESS` с собственными ограничениями и применить его для столбца вместо типа `VARCHAR`. Специальным ограничением столбца в базе данных SQL является `NOT NULL`;
- *табличные ограничения* (table constraints). Правило целостности, применяемое к нескольким столбцам или нескольким строкам, является табличным ограничением. Типичным декларативным табличным ограничением является `UNIQUE`: каждая строка проверяется на наличие повторяющихся значений (например, все пользователи должны иметь различные электронные адреса). Примером может служить правило, влияющее лишь на одну запись, но на несколько столбцов, которое гласит: «Время окончания аукциона должно быть позже времени его начала»;
- *ограничение базы данных* (database constraints). Если правило применяется более чем к одной таблице, оно действует на уровне всей базы данных. Вы наверняка уже знакомы с самым распространенным ограничением базы данных – внешним ключом. Это правило гарантирует ссылочную целостность между записями, как правило, но не всегда, расположенными в разных таблицах (рекурсивные ограничения внешнего ключа встречаются не так уж редко). Также нередко встречаются другие ограничения базы данных, включающие несколько таблиц. Например, ставка за товар может сохраняться, только если еще не наступило время окончания аукциона для данного товара.

Большинство (если не все) СУБД SQL поддерживает данные виды ограничений, а также наиболее важные их разновидности. В дополнение к таким ключевым словам, как `NOT NULL` и `UNIQUE`, обычно можно объявлять более сложные правила, используя ограничение `CHECK` с произвольным выражением SQL. Но ограничения целостности по-прежнему являются слабым местом стандарта SQL, и решения, предлагаемые разработчиками СУБД, могут значительно отличаться.

Кроме того, с помощью триггеров в базе данных, перехватывающих операции изменения данных, можно создавать недекларативные и процедурные ограничения. Триггер может реализовывать процедуру проверки ограничения непосредственно или вызывать существующую хранимую процедуру.

Ограничения целостности могут проверяться сразу же, во время выполнения инструкций, изменяющих данные, или откладываться до окончания транзакции. Обычно на нарушение правил базы данных SQL отвечают отказом, без возможности изменения такого поведения. Внешние ключи занимают особое место, поскольку вы, как правило, можете определить, что должно происходить с зависимыми строками при удалении (`ON DELETE`) или изменении (`ON UPDATE`).

Hibernate представляет нарушение ограничения базы данных в виде объекта исключения; проверьте, не имеется ли в цепочке исключений объекта типа `org.hibernate.exception.ConstraintViolationException`. Это исключение может сообщить больше информации об ошибке, например имя нарушенного ограничения базы данных.

Вывод сообщений об ошибках валидации

Это слишком хорошо, чтобы быть правдой: если уровень взаимодействия с базой данных возбудит исключение типа `ConstraintViolationException`, то почему бы не показать его пользователю? Пользователь сможет затем менять неправильное имя на своем экране и отправлять форму заново до тех пор, пока не пройдет валидацию. К сожалению, это так не работает. Многие, кто пытался реализовать эту стратегию, терпели поражение.

Во-первых, каждая СУБД определяет свои сообщения об ошибках, и Hibernate не гарантирует корректного их анализа. Информация, предоставляемая объектом `ConstraintViolationException`, – это всего лишь лучшая из догадок; обычно она неверна и подходит лишь для создателей сообщений, записываемых в журнал. Должно ли это быть стандартизировано в SQL? Безусловно; но этого не происходит.

Во-вторых, приложение не должно передавать базе данных некорректную информацию, проверяя, что подойдет, а что нет. СУБД – это последний рубеж обороны, а не первая линия валидации. Вместо этого используйте на уровне приложения Java механизм Bean Validation и возвращайте своим пользователям понятные сообщения об ошибках на их родном языке.

Давайте поближе рассмотрим реализацию ограничений целостности.

Добавление ограничений доменов и столбцов

Стандарт SQL определяет домены, которые, к сожалению, достаточно ограничены и не всегда поддерживаются СУБД. Если ваша система поддерживает домены SQL, можете воспользоваться ими для ограничнения типов данных.

Определим в SQL-сценарии создания домен `EMAIL_ADDRESS` на основе типа `VARCHAR`:

Файл: /model/src/main/resources/complexschemas/CreateScript.sql.txt

```
create domain if not exists
  EMAIL_ADDRESS as varchar
  check (position('@', value) > 1);
```

Дополнительным ограничением является проверка наличия символа @ в строке. Преимущество (относительно небольшое) такого домена в том, что общие ограничения находятся в одном месте. Ограничения домена всегда проверяются сразу же во время вставки и изменения данных.

Теперь вы можете применять этот домен в своих отображениях, как встроенный тип данных:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```
@Entity
public class User {

    @Column(
        nullable = false,   ← Ограничение столбца
        unique = true,      ← Табличное ограничение для нескольких строк
        columnDefinition = "EMAIL_ADDRESS(255)"   ← Применение ограничения домена
    )
    protected String email;

    // ...
}
```

В этом отображении задействовано несколько ограничений. Ограничение `NOT NULL` применяется повсеместно; вы уже многократно с ним встречались. Вторым следует ограничение столбца `UNIQUE`; никакие два пользователя не могут иметь одинаковых электронных адресов. На момент написания книги в Hibernate нельзя было поменять имя ограничения уникальности для одного столбца; в схеме оно получало нечитаемое, автоматически сгенерированное имя. Последним идет параметр `columnDefinition`, ссылающийся на домен, добавленный выше в сценарии создания. Данное определение является фрагментом SQL, который экспортируется в схему напрямую, поэтому будьте внимательны с кодом SQL, зависящим от конкретной базы данных.

Если нет желания определять домены заранее, используйте в качестве ограничения для одного столбца ключевое слово `CHECK`:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```
@Entity
public class User {
    @Column(columnDefinition =
        "varchar(15) not null unique" +
        " check (not substring(lower(USERNAME), 0, 5) = 'admin')"
    )
```

```
    protected String username;  ← Аннотация @org.hibernate.annotations.Check
                                  пока не поддерживается для свойств классов
    // ...
}
```

Данное определение ограничивает длину корректного имени пользователя 15 символами; также, во избежание путаницы, строка не может начинаться со слова *admin*. Можно вызывать любую функцию SQL, поддерживаемую базой данных; значение параметра `columnDefinition` всегда помещается в экспортируемую схему.

Обратите внимание, что вам предоставлен выбор: создание и применение домена окажут такой же эффект, как добавление ограничения для одного столбца. Как правило, домены легче поддерживать, и они помогают избежать дублирования кода.

На момент написания этих строк Hibernate не поддерживал аннотацию `@org.hibernate.annotations.Check` для отдельных свойств класса; используйте ее для табличных ограничений.

Особенности Hibernate

Ограничения уровня таблицы

Аукцион не может завершиться до своего начала. До сих пор ни в схеме SQL, ни в предметной модели не было правила, реализующего данное ограничение. Нам понадобится табличное ограничение для одной строки:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/custom/Item.java

```
@Entity
@org.hibernate.annotations.Check(
    constraints = "AUCTIONSTART < AUCTIONEND"
)
public class Item {

    @NotNull
    protected Date auctionStart;

    @NotNull
    protected Date auctionEnd;

    // ...
}
```

Hibernate добавляет табличные ограничения, которые могут содержать произвольные выражения SQL, в сформированную инструкцию `CREATE TABLE`.

С помощью более сложных выражений можно реализовать многострочные табличные ограничения. Для этой цели могут понадобиться подзапросы, которые могут не поддерживаться вашей СУБД. Однако существуют распространенные многострочные табличные ограничения, такие как `UNIQUE`, которые можно ис-

пользовать прямо в отображениях. Вы уже видели аннотацию `@Column(unique = true|false)` в предыдущем разделе.

Если ограничение уникальности распространяется на несколько столбцов, используйте параметр `uniqueConstraints` аннотации `@Table`:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```
@Entity
@Table(
    name = "USERS",
    uniqueConstraints =
        @UniqueConstraint(
            name = "UNQ_USERNAME_EMAIL",
            columnNames = { "USERNAME", "EMAIL" }
        )
)
public class User {

    // ...
}
```

Теперь каждая пара значений `USERNAME` и `EMAIL` должна быть уникальной для всех записей в таблице `USERS`. Если не указать имя ограничения – как в данном случае `UNQ_USERNAME_EMAIL`, оно будет сгенерировано автоматически и, возможно, будет не таким удобочитаемым.

Последний вид ограничений, который мы рассмотрим, – ограничения базы данных, распространяющиеся на несколько таблиц.

Ограничения уровня базы данных

Пользователи могут делать ставки, пока аукцион не завершится. База данных должна гарантировать сохранение только корректных ставок, чтобы при каждой вставке строки в таблицу `BID` метка времени `CREATEDON` (когда ставка была создана) сверялась со временем окончания аукциона. Данный тип ограничения затрагивает две таблицы: `BID` и `ITEM`.

В любом SQL-выражении `CHECK` можно создать правило, распространяющееся на несколько таблиц, используя соединение с подзапросом. Вместо того чтобы ссылаться только на одну таблицу, в которой объявлено ограничение, можно написать запрос (как правило, проверяющий наличие/отсутствие тех или иных данных) к другой таблице. Но проблема в том, что вы не можете использовать аннотацию `@org.hibernate.annotations.Check` ни в классе `Bid`, ни в классе `Item`. Потому что неизвестно, какую таблицу Hibernate создаст первой.

Поэтому ограничение `CHECK` следует поместить в выражение `ALTER TABLE`, выполняемое после создания всех таблиц. Для этого хорошо подходит сценарий загрузки, поскольку он выполняется как раз в этот момент:

Файл: /model/src/main/resources/complexschemas/LoadScript.sql.txt

```
alter table BID
  add constraint AUCTION_BID_TIME
  check(
    CREATEDON <= (
      select i.AUCTIONEND from ITEM i where i.ID = ITEM_ID
    )
  );
```

Запись в таблице BID теперь будет считаться корректной, только если значение CREATEDON в ней будет меньше или равно значению времени окончания аукциона в соответствующей записи таблицы ITEM.

Большинство встречавшихся нам до сих пор правил, распространяющихся на несколько таблиц, было правилами ссылочной целостности. Они широко известны как внешние ключи, представляющие собой комбинацию двух элементов: копии ключа связанной строки и ограничения, гарантирующего существование связанного значения. Hibernate автоматически создаст ограничение внешнего ключа для каждого столбца внешнего ключа в отображении связи. Если посмотреть на схему, сформированную Hibernate, вы заметите, что для этих ограничений автоматически созданы идентификаторы в базе данных, которые не очень легко читаются, что может затруднять отладку. В сгенерированной схеме можно, например, увидеть выражения такого вида:

```
alter table BID add constraint FKCFAEEDB471BF59FF
    foreign key (ITEM_ID) references ITEM
```

Это выражение определяет ограничение внешнего ключа для столбца ITEM_ID в таблице BID, ссылающегося на столбец первичного ключа в таблице ITEM. Вы можете поменять имя ограничения с помощью параметра foreignKey отображения @JoinColumn:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/custom/Bid.java

```
@Entity
public class Bid {

    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        nullable = false,
        foreignKey = @ForeignKey(name = "FK_ITEM_ID")
    )
    protected Item item;

    // ...
}
```

Атрибут foreignKey также поддерживается отображениями @PrimaryKeyJoinColumn, @MapKeyJoinColumn, @JoinTable, @CollectionTable и @AssociationOverride.

Аннотация `@ForeignKey` имеет несколько редко используемых параметров, не показанных ранее:

- вы можете задать собственное определение внешнего ключа, присвоив параметру `foreignKeyDefinition` фрагмент SQL-кода в формате `FOREIGN KEY ([column]) REFERENCES [table]([column]) ON UPDATE [action]`. Hibernate будет использовать этот фрагмент вместо того, который сформирует реализация механизма хранения; данный фрагмент может быть написан на диалекте SQL, поддерживаемом вашей СУБД;
- параметр `ConstraintMode` со значением `NO_CONSTRAINT` подойдет в случаях, когда требуется полностью отказаться от создания внешнего ключа. Вы можете написать ограничение внешнего ключа самостоятельно, используя выражение `ALTER TABLE`, например поместив его в сценарий загрузки, как было показано ранее.

Правильное именование ограничений не только является хорошим тоном, но также может помочь при анализе сообщений об исключениях.

На этом мы завершаем обсуждение правил целостности в базе данных. Далее мы рассмотрим некоторые оптимизации, которые вам, возможно, захочется добавить в свою схему для улучшения производительности.

9.1.3. Создание индексов

Индексы – это ключевой аспект оптимизации производительности приложения, работающего с базой данных. Оптимизатор запросов в СУБД может воспользоваться индексом, чтобы избежать избыточного сканирования таблиц данных. Поскольку индексы относятся к физической реализации базы данных, они не являются частью стандарта SQL, а синтаксис DDL и доступные варианты индексирования зависят от конкретного продукта. Тем не менее вы можете поместить в метаданные отображения наиболее общие артефакты схемы для типичных индексов.

Многие запросы в приложении CaveatEmptor почти наверняка будут обращаться к свойству `username` сущности `User`. Вы можете ускорить эти запросы, проиндексировав соответствующий столбец. Другим кандидатом на индексацию является комбинация столбцов `USERNAME` и `EMAIL`, которые также могут часто использоваться в запросах. Объявить индексы для одного или нескольких столбцов можно в атрибуте `indexes` аннотации `@Table` перед классом сущности.

Файл: /model/src/main/java/org/jpwh/model/complexschemas/custom/User.java

```
@Entity
@Table(
    name = "USERS",
    indexes = {
        @Index(
            name = "IDX_USERNAME",
            columnList = "USERNAME"
        ),
```

```
        @Index(
            name = "IDX_USERNAME_EMAIL",
            columnList = "USERNAME, EMAIL"
        )
    }
)
public class User {

    // ...
}
```

Если имя индекса не указано, будет использоваться сгенерированное имя.

Мы не советуем добавлять индекс, если вам только кажется, будто он сможет решить проблему производительности. Прочтите отличную книгу *SQL Tuning* Дэна Тоу (Tow, 2003), где вы познакомитесь с эффективными приемами оптимизации и особенно с тем, как индексы помогают приблизиться к самому производительному плану выполнения запросов.

Настройка схемы базы данных обычно возможна, только если вы работаете с новой системой, не содержащей данных. При работе с существующей унаследованной схемой часто возникает проблема взаимодействия с естественными и составными ключами.

9.2. Унаследованные первичные ключи

В разделе 4.2.3 мы упоминали, что считаем использование естественных первичных ключей плохой идеей. Естественные ключи затрудняют внесение изменений в модель данных при изменении бизнес-требований. Иногда они могут даже снижать производительность. К сожалению, во многих унаследованных схемах интенсивно используются (естественные) составные ключи; и как раз по причине, по которой мы не поддерживаем их применения, вам может оказаться трудно поменять старую схему, чтобы использовать естественные несоставные или суррогатные ключи. По этой причине JPA поддерживает естественные ключи, а также составные первичные и внешние ключи.

9.2.1. Отображение естественных первичных ключей

Если в унаследованной схеме вы встретите таблицу `USERS`, вероятнее всего, ее первичным ключом будет столбец `USERNAME`. В таком случае у вас не будет суррогатного идентификатора, который Hibernate мог бы сгенерировать автоматически. Вместо этого само приложение должно присваивать значение идентификатора при сохранении экземпляра класса `User`:

Файл: /examples/src/test/java/org/jpwh/test/complexschemas/NaturalPrimaryKey.java

```
User user = new User("johndoe");
em.persist(user);
```

Здесь единственным аргументом общедоступного конструктора класса `User` является имя пользователя:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/naturalprimarykey/ User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @Id
    protected String username;

    protected User() {
    }

    public User(String username) {
        this.username = username;
    }

    // ...
}
```

При загрузке экземпляра `User` из базы данных Hibernate вызовет защищенный конструктор без аргументов, а затем напрямую присвоит значение полю `username`. Создавая экземпляр класса `User`, вызовите общедоступный конструктор с аргументом `username`. Если вы не объявили генератор идентификаторов перед свойством с аннотацией `@Id`, Hibernate будет полагать, что приложение само позаботится о присваивании первичного ключа.

Составные (естественные) первичные ключи требуют больших затрат.

9.2.2. Отображение составных первичных ключей

Предположим, что первичный ключ таблицы `USERS` включает два столбца: `USERNAME` и `DEPARTMENTNR`. Тогда нужно написать отдельный класс составного идентификатора, объявляющего только ключевые поля, и назвать его `UserId`:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/embedded/UserId.java

```
@Embeddable   ← ❶ Встраиваемый (@Embeddable) и сериализуемый (Serializable) класс
public class UserId implements Serializable {

    protected String username;   ← ❷ Автоматически генерируется ограничение NOT NULL

    protected String departmentNr;

    protected UserId() {   ← ❸ Защищенный конструктор
    }

    public UserId(String username, String departmentNr) {   ← ❹ Общедоступный конструктор
        this.username = username;
        this.departmentNr = departmentNr;
    }
```

```
    @Override  ◄— ❺ Переопределение equals() и hashCode()
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        UserId userId = (UserId) o;
        if (!departmentNr.equals(userId.departmentNr)) return false;
        if (!username.equals(userId.username)) return false;
        return true;
    }

    @Override  ◄— ❺ Переопределение equals() и hashCode()
    public int hashCode() {
        int result = username.hashCode();
        result = 31 * result + departmentNr.hashCode();
        return result;
    }

    // ...
}
```

❶ Этот класс должен быть встраиваемым (`@Embeddable`) и сериализуемым (`Serializable`) – любой тип, используемый в JPA как идентификатор, должен быть сериализуемым (`Serializable`).

❷ Не нужно отмечать свойства составного ключа аннотациями `@NotNull`; соответствующие им столбцы в базе данных автоматически получат ограничение `NOT NULL`, поскольку будут входить в состав первичного ключа сущности.

❸ Спецификация JPA требует наличия общедоступного конструктора без аргументов у встраиваемого класса идентификатора. Hibernate допускает защищенную область видимости.

❹ Единственный общедоступный конструктор должен получать значения ключа в виде аргументов.

❺ Вы должны переопределить методы `equals()` и `hashCode()`, сохраняя ту же семантику составного ключа, что и в базе данных. В данном случае это простое сравнение значений `username` и `departmentNr`.

Все это – важные части класса `UserId`. Возможно, вы добавите несколько методов чтения для получения значений свойств.

Теперь осталось отобразить сущность `User`, используя этот тип идентификатора и аннотацию `@EmbeddedId`:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/embedded/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @EmbeddedId
    protected UserId id;  ◄— Можно добавить аннотацию @AttributeOverrides

    public User(UserId id) {
```

```
        this.id = id;
    }

    // ...
}
```

Так же, как для обычных встроенных компонентов, можно переопределять отдельные атрибуты и отображаемые ими столбцы, как было показано в разделе 5.2.3. Схема базы данных показана на рис. 9.1.

Любой общедоступный конструктор класса `User` должен требовать передачи экземпляра `UserId`, вынуждая определять значение перед сохранением объекта `User` (также в классе сущности должен быть определен конструктор без аргументов):

Файл: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyEmbeddedId.java

```
UserId id = new UserId("johndoe", "123");
User user = new User(id);
em.persist(user);
```

Вот так вы можете загрузить экземпляр класса `User`:

Файл: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyEmbeddedId.java

```
UserId id = new UserId("johndoe", "123");
User user = em.find(User.class, id);
assertEquals(user.getId().getDepartmentNr(), "123");
```

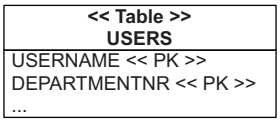


Рис. 9.1 ❖ В таблице `USERS` определен составной первичный ключ

Далее предположим, что столбец `DEPARTMENTNR` – это внешний ключ, ссылающийся на таблицу `DEPARTMENT`, и требуется представить эту связь в предметной модели на Java в виде связи *многие к одному*.

9.2.3. Внешние ключи внутри составных первичных ключей

Взгляните на схему на рис. 9.2.

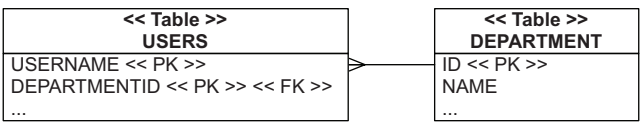


Рис. 9.2 ❖ Часть составного первичного ключа в таблице `USERS` также является внешним ключом

Сначала попробуем реализовать отображение с помощью аннотации `@MapsId`, разработанной специально для этой цели. Начнем с переименования поля `departmentNr` в `departmentId` во встроенном классе идентификатора `UserId`, показанного в предыдущем разделе:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/mapsid/UserId.java

```
@Embeddable
public class UserId implements Serializable {

    protected String username;

    protected Long departmentId;

    // ...
}
```

Теперь свойство имеет тип `Long`, а не `String`. Далее в классе сущности `User` объявим связь с помощью отображения `@ManyToOne` перед полем `department`:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/mapsid/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @EmbeddedId
    protected UserId id;

    @ManyToOne
    @MapsId("departmentId")
    protected Department department;

    public User(UserId id) {
        this.id = id;
    }

    // ...
}
```

Аннотация `@MapsId` сообщает фреймворку Hibernate, что при сохранении экземпляра `User` значение свойства `UserId#departmentId` должно игнорироваться. Вместо этого при сохранении записи в таблицу `USERS` Hibernate возьмет идентификатор объекта `Department` из свойства `User#department`:

Файл: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyMapsId.java

```
Department department = new Department("Sales");
em.persist(department);

UserId id = new UserId("johndoe", null);  ← Null?
User user = new User(id);
```

```
user.setDepartment(department);  ← Обязательно
em.persist(user);
```

При сохранении Hibernate игнорирует любое значение, присвоенное `UserId#departmentId`; даже `null`, присвоенное в данном примере. Это означает, что перед сохранением объекта `User` у вас уже должен иметься экземпляр `Department`. В JPA это называется косвенным отображением идентификатора.

При загрузке объекта `User` необходим только идентификатор объекта `Department`:

Файл: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyMapsId.java

```
UserId id = new UserId("johndoe", DEPARTMENT_ID);
User user = em.find(User.class, id);
assertEquals(user.getDepartment().getName(), "Sales");
```

Нам не очень нравится эта стратегия отображения. Вот способ получше, не требующий аннотации `@MapsId`:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/readonly/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @EmbeddedId
    protected UserId id;

    @ManyToOne
    @JoinColumn(
        name = "DEPARTMENTID",  ← По умолчанию DEPARTMENT_ID
        insertable = false, updatable = false  ← Доступ только для чтения
    )
    protected Department department;

    public User(UserId id) {
        this.id = id;
    }

    // ...
}
```

С помощью обычных параметров `insertable=false` и `updatable=false` мы разрешаем доступ к свойству `User#department` только для чтения. Это значит, что можно лишь запрашивать данные, делая вызов `someUser.getDepartment()`, а общедоступного метода `setDepartment()` просто нет. Теперь за изменение столбца `DEPARTMENTID` в таблице `USERS` отвечает свойство `UserId#departmentId`.

И перед сохранением нового пользователя (`User`) мы должны установить значение идентификатора подразделения:

Файл: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyReadOnly.java

```
Department department = new Department("Sales");
em.persist(department);  ← Присваивает первичный ключ

UserId id = new UserId("johndoe", department.getId());  ← Обязательно
User user = new User(id);
em.persist(user);

assertNull(user.getDepartment());  ← Осторожно!
```

Обратите внимание, что метод `User#getDepartment()` возвращает значение `null`, потому что мы не присвоили значения этому свойству. Hibernate заполнит его только при загрузке экземпляра `User`:

Файл: /examples/src/test/java/org/jpwh/test/complexschemas/CompositeKeyReadOnly.java

```
UserId id = new UserId("johndoe", DEPARTMENT_ID);
User user = em.find(User.class, id);
assertEquals(user.getDepartment().getName(), "Sales");
```

Многие разработчики предпочитают инкапсулировать эти подробности в конструкторе:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/readonly/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    public User(String username, Department department) {
        if (department.getId() == null)
            throw new IllegalStateException(
                "Department is transient: " + department
            );
        this.id = new UserId(username, department.getId());
        this.department = department;
    }

    // ...
}
```

Такой конструктор гарантирует правильную инициализацию объекта `User`, корректно устанавливая значения идентификатора и свойств.

Если в таблице `USERS` определен составной первичный ключ, внешний ключ, ссылающийся на нее, также должен быть составным.

9.2.4. Внешний ключ, ссылающийся на составной первичный ключ

Например, связь от класса `Item` к классу `User`, представленная свойством `seller`, может потребовать отображения составного внешнего ключа. Посмотрите на схему на рис. 9.3.

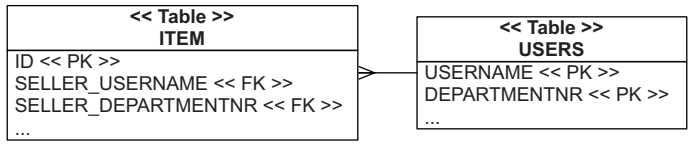


Рис. 9.3 ❖ Продавец представлен в таблице `ITEM` с помощью составного внешнего ключа

Hibernate может спрятать эти подробности внутри предметной модели на Java. Например, вот как выглядит отображение поля `Item#seller`:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/compositekey/manytoone/Item.java

```
@Entity
public class Item {
@NotNull

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name = "SELLER_USERNAME",
                    referencedColumnName = "USERNAME"),
        @JoinColumn(name = "SELLER_DEPARTMENTNR",
                    referencedColumnName = "DEPARTMENTNR")
    })
    protected User seller;

    // ...
}
```

Возможно, вам не приходилось встречать аннотацию `@JoinColumns` раньше; она определяет список столбцов составного внешнего ключа, реализующего данную связь. Чтобы связать источник с целью внешнего ключа, обязательно нужно определить атрибут `referencedColumnName`. К сожалению, Hibernate ничего не сообщит, если не сделать этого, в результате можно получить некорректный порядок столбцов в сгенерированной схеме.

В унаследованных схемах внешние ключи не всегда ссылаются на первичные ключи.

9.2.5. Внешние ключи, ссылающиеся на непервичные ключи

Ограничение внешнего ключа столбца SELLER в таблице ITEM гарантирует существование продавца товара, требуя, чтобы один продавец был представлен в *некотором* столбце *некоторой* записи *некоторой* таблицы. Других правил нет; целевой столбец должен быть первичным ключом или иметь ограничение уникальности. В качестве целевой может выступать любая таблица. Значение может быть числовым идентификатором продавца или строкой с номером клиента; единственное требование: тип столбца внешнего ключа должен совпадать с типом целевого столбца, на который он ссылается.

Конечно, ограничение внешнего ключа обычно ссылается на столбец (столбцы) первичного ключа. Тем не менее в унаследованных базах данных иногда можно найти такие ограничения внешнего ключа, которые не следуют этому простому правилу. Иногда ограничение внешнего ключа может ссылаться на столбец с ограничением уникальности – естественный непервичный ключ. Предположим, что в приложении CaveatEmptor требуется поддержка унаследованного старого столбца естественного ключа CUSTOMERNR в таблице USERS, как показано на рис. 9.4:

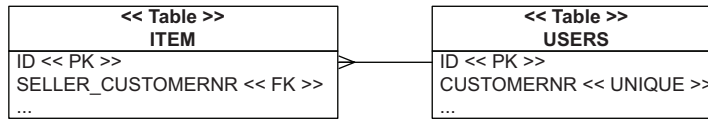


Рис. 9.4 ❖ Внешний ключ, представляющий продавца, ссылается на неключевой столбец таблицы USERS

Файл: /model/src/main/java/org/jpwh/model/complexschemas/naturalforeignkey/ User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    @Column(unique = true)
    protected String customerNr;

    // ...
}
```

Пока ничего необычного; вы уже видели подобное простое отображение уникального поля. Аспектом унаследованной схемы будет столбец SELLER_CUSTOMERNR таблицы ITEM с ограничением внешнего ключа, ссылающимся на столбец CUSTOMERNR в таблице USERS, а не на ID:

Файл: /model/src/main/java/org/jpwh/model/complexschemas/naturalforeignkey/ Item.java

```
@Entity
public class Item {

    @NotNull
    @ManyToOne
    @JoinColumn(
        name = "SELLER_CUSTOMERNR",
        referencedColumnName = "CUSTOMERNR"
    )
    protected User seller;

    // ...
}
```

Для объявления этого отношения нужно указать значение атрибута `referencedColumnName` в аннотации `@JoinColumn`. Теперь Hibernate будет знать, что целевой столбец является естественным, а не первичным ключом, и управлять отношением внешнего ключа соответственно.

Если целевой естественный ключ является составным, используйте аннотацию `@JoinColumns`, как в предыдущем разделе. К счастью, такую схему всегда можно просто привести в порядок, выполнив рефакторинг внешних ключей, чтобы они ссылались на первичные ключи, если только у вас есть возможность вносить изменения в базу данных, не мешая другим приложениям, использующим эти данные.

На этом мы завершаем обсуждение проблем естественных, составных и внешних ключей, с которыми вы можете столкнуться при отображении унаследованной схемы. Давайте перейдем к другой интересной специальной стратегии: отображению свойств сущности простого или встраиваемого типа во вторичную таблицу.

9.3. Отображение свойств во вторичные таблицы

Мы уже показывали аннотацию `@SecondaryTable` в отображении наследования, в разделе 6.5. Она помогла нам вынести поля подкласса в отдельную таблицу. Эта обобщенная функциональность имеет множество применений, но помните, что число классов в системе должно быть больше, чем таблиц.

Представьте, что в унаследованной схеме платежный адрес пользователя хранится в отдельной таблице, а не вместе с остальной информацией в главной таблице сущности `USERS`. Эта схема показана на рис. 9.5. Домашний адрес пользователя хранится в столбцах `STREET`, `ZIPCODE` и `CITY` таблицы `USERS`. Платежный адрес пользователя хранится в таблице `BILLING_ADDRESS` со столбцом первичного ключа `USER_ID`, имеющим также ограничение внешнего ключа, ссылающегося на столбец первичного ключа `ID` в таблице `USERS`.

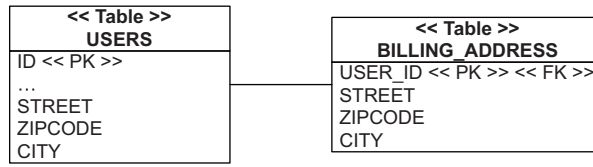


Рис. 9.5 ❖ Хранение данных о платежном адресе в отдельной таблице

Для отображения этой схемы объявим вторичную таблицу для сущности `User` и определим, как Hibernate должен соединить ее со вторичной таблицей (`@SecondaryTable`):

Файл: /model/src/main/java/org/jpwh/model/complexschemas/secondarytable/User.java

```
@Entity
@Table(name = "USERS")
@SecondaryTable(
    name = "BILLING_ADDRESS",
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "USER_ID")
)
public class User {

    protected Address homeAddress;

    @AttributeOverrides({
        @AttributeOverride(name = "street",
            column = @Column(table = "BILLING_ADDRESS",
                             nullable = false)),
        @AttributeOverride(name = "zipcode",
            column = @Column(table = "BILLING_ADDRESS",
                             length = 5,
                             nullable = false)),
        @AttributeOverride(name = "city",
            column = @Column(table = "BILLING_ADDRESS",
                             nullable = false))
    })
    protected Address billingAddress;

    // ...
}
```

В классе `User` объявлены два свойства встраиваемого типа: `homeAddress` и `billingAddress`. Первое представляет обычное встроенное отображение, а класс `Address` является встраиваемым (`@Embeddable`).

Так же как в разделе 5.2.3, для переопределения отображений встраиваемых свойств здесь можно использовать аннотацию `@AttributeOverrides`. Аннотация `@Column` отображает отдельные свойства в таблицу `BILLING_ADDRESS` с помощью параметра `table`. Помните, что аннотация `@AttributeOverride` замещает всю инфор-

мацию об отображении свойства: любые аннотации перед свойствами класса `Address` будут игнорироваться при переопределении. Поэтому мы должны заново указать возможность присваивания null и длину поля, используя переопределяющую аннотацию `@Column`.

Мы показали вам отображение вторичной таблицы на примере встраиваемых полей. Вы могли бы также вынести во вторичную таблицу свойства простых типов, такие как `username`. Но помните, что чтение и сопровождение этих отображений могут превратиться в проблему; использовать вторичные таблицы для отображения следует только в унаследованных неизменяемых старых схемах.

9.4. Резюме

- Сосредоточьтесь на схеме базы данных.
- Вы можете добавлять дополнительные ограничения целостности в схему базы данных, сгенерированную Hibernate. Теперь вы знаете, как выполнять произвольные SQL-сценарии создания, удаления и загрузки.
- Мы обсудили применение SQL-ограничений доменов, столбцов, таблиц и базы данных.
- Также мы обсудили применение пользовательских SQL-типов данных и ограничений уникальности, произвольной проверки и внешнего ключа.
- Вы познакомились с некоторыми типичными проблемами, которые приходится решать при работе с унаследованными схемами и особенно ключами.
- Вы узнали о нескольких типах отображений: естественных первичных ключах, составных первичных ключах, внешних ключах внутри составных первичных ключей, внешних ключах, ссылающихся на составные первичные ключи, и внешних ключах, ссылающихся на непервичные ключи.
- Вы узнали, как переместить свойства сущности во вторичную таблицу.

Часть III

ТРАНЗАКЦИОННАЯ ОБРАБОТКА ДАННЫХ

В части III мы займемся загрузкой и сохранением данных с применением Hibernate и Java Persistence. Мы рассмотрим программные интерфейсы, способы создания транзакционных приложений и как Hibernate может обеспечить более эффективную загрузку данных.

Начиная с главы 10, вы приступите к изучению наиболее важных стратегий работы с экземплярами сущностей в JPA-приложении. Познакомитесь с жизненным циклом экземпляров сущностей и увидите, как они сохраняются, отсоединяются и удаляются. Именно в этой главе вы познакомитесь с наиболее важным JPA-интерфейсом: `EntityManager`. Далее, в главе 11, будут описаны основы системных транзакций и транзакций базы данных, а также управление параллельным доступом с помощью Hibernate и JPA. Вы также познакомитесь с нетранзакционным доступом к данным. В главе 12 мы рассмотрим отложенную и немедленную загрузки, планы извлечения, стратегии и профили, а закончим оптимизацией выполнения кода SQL. Наконец, в главе 13 будут рассмотрены каскадная передача изменений состояния, ожидание и перехват событий, аудит и версионирование с помощью Hibernate Envers, а также динамическая фильтрация данных.

К концу этой части вы будете знать, как с помощью программных интерфейсов Hibernate и Java Persistence наиболее эффективно загружать, изменять и сохранять объекты. Вы поймете, как работают транзакции и почему диалоговые взаимодействия могут открыть новые подходы к проектированию приложений. Вы будете готовы к оптимизации любого сценария модификации объектов и к применению эффективных стратегий извлечения и кэширования для увеличения производительности и масштабирования.

Глава 10

Управление данными

В этой главе:
- жизненный цикл и состояния объектов;
- работа с Java Persistence API;
- работа с отсоединенным состоянием.

Вы уже знаете, как Hibernate и ORM работают со статическими аспектами объектно-реляционного рассогласования. Используя знания, полученные к настоящему моменту, вы можете создавать отображения между Java-классами и SQL-схемами, разрешая проблемы структурного несоответствия. Чтобы освежить в памяти сведения о разрешении таких проблем, можно еще раз просмотреть раздел 1.2.

Но для достижения высокой эффективности разрабатываемого приложения этого недостаточно: требуется еще изучить стратегии управления данными во время выполнения. Эти стратегии имеют решающее значение для высокой производительности и правильной работы приложений.

Данная глава будет посвящена жизненному циклу экземпляров сущностей, процессу перехода их в хранимое состояние и процессу выхода их из хранимого состояния, а также методам и операциям, осуществляющим эти переходы. `EntityManager` является основным интерфейсом доступа к данным в JPA.

Прежде чем перейти к исследованию API, разберемся с экземплярами объектов, их жизненным циклом и событиями, приводящими к изменению их состояния. Хотя часть сведений является формальной, ясное понимание жизненного цикла хранения имеет важное значение.

Главные нововведения в JPA 2

- Получить конкретную реализацию диспетчера хранения можно вызовом метода `EntityManager#unwrap()`: например, API `org.hibernate.Session`. Используйте метод `EntityManagerFactory#unwrap()` для получения экземпляра `org.hibernate.SessionFactory`.
- Новая операция `detach()` обеспечивает точное управление контекстом хранения, вплоть до отсоединения отдельных экземпляров сущностей.
- Из существующего объекта `EntityManager` можно получить объект `EntityManagerFactory`, используемый для создания контекста хранения с помощью `getEntityManagerFactory()`.

- Новые статические методы `Persistence(Unit)Util` позволяют определить, был ли экземпляр объекта (или одно из его свойств) полностью загружен или является неинициализированной ссылкой (прокси-объектом Hibernate или оберткой выгруженной коллекции).

10.1. Жизненный цикл хранения

Поскольку JPA является механизмом прозрачного хранения, классы ничего не знают о возможности собственного сохранения; можно написать такую логику приложения, которой будет неизвестно, представляют ли данные, с которыми она работает, хранимое или временное состояние, существующее только в памяти. При вызове методов экземпляра приложение не должно беспокоиться о том, является ли оно хранимым. Вы можете, к примеру, вызвать бизнес-метод `Item#calculateTotalPrice()`, вообще не задумываясь о хранении (например, в модульном тесте).

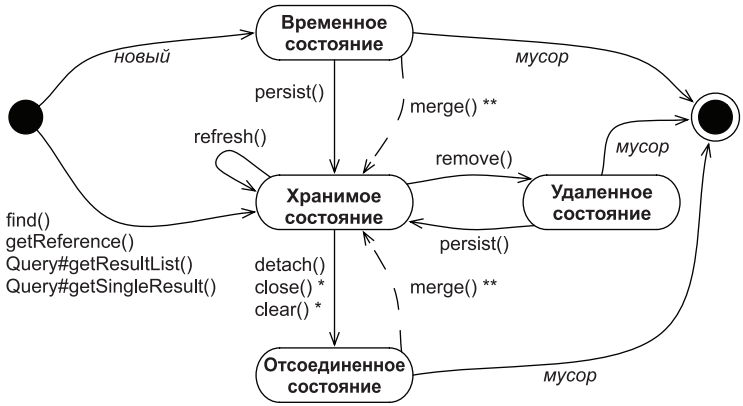
Любое приложение с хранимым состоянием должно взаимодействовать со службой хранения, когда требуется перенести текущее состояние из памяти в базу данных (и наоборот). Иначе говоря, для сохранения и загрузки данных вы должны вызывать интерфейсы Java Persistence.

При такой организации работы с механизмом хранения приложение должно учитывать не только факт хранения в базе данных, но и жизненный цикл экземпляра сущности. Мы называем его *жизненным циклом хранения*, представляющим набор состояний, через которые проходит экземпляр сущности за время своего существования. Также мы будем использовать термин *единица работы* – набор операций, меняющих (возможно) состояние, которые выполняются (обычно атомарно) как единое целое. Другой частью головоломки является *контекст хранения*, предоставляемый службой хранения. Вы можете рассматривать контекст хранения как службу, запоминающую все модификации и изменения состояния, которые производятся над данными в рамках конкретной единицы работы (это несколько упрощенное представление, но для начала вполне уместное).

Теперь разберемся со всеми этими терминами: состояниями сущностей, контекстами хранения и управляемыми областями видимости. Вы, возможно, больше привыкли думать о тех инструкциях SQL, которые нужно использовать для загрузки и сохранения данных в базу; но одним из решающих факторов для успешной работы с Java Persistence является понимание процесса *управления состоянием*, поэтому задержитесь в этом разделе вместе с нами.

10.1.1. Состояния экземпляров сущностей

Различные механизмы ORM используют разную терминологию и определяют различные состояния и переходы между ними внутри персистентного жизненного цикла. Более того, внутренние состояния могут отличаться от состояний, предоставляемых клиентскому приложению. JPA определяет четыре состояния, скрывая сложность внутренней реализации Hibernate от клиентского кода. Эти состояния и переходы между ними показаны на рис. 10.1.



* Влияет на все экземпляры внутри контекста хранения
** Операция слияния возвращает хранимый экземпляр; оригинальный экземпляр при этом не меняет своего состояния

Рис. 10.1 ❖ Состояния экземпляров сущностей и переходы между ними

На диаграмме состояний также показаны вызовы методов интерфейса `EntityManager` (и `Query`), инициирующие переходы между состояниями. Мы обсудим данную диаграмму в этой главе; вы всегда можете обратиться к ней для получения общей картины. Давайте более детально изучим состояния и переходы между ними.

Временное состояние

Объекты, созданные с помощью Java-оператора `new`, являются *временными* (transient), т. е. их состояние будет потеряно и уничтожено сборщиком мусора, как только исчезнут последние ссылки на них. Например, выражение `new Item()` создает временный экземпляр класса `Item`, подобно `new Long()` и `new BigDecimal()`. Hibernate не поддерживает возможности отката изменений во временных экземплярах; изменив цену временного товара (`Item`), вы не сможете автоматически отменить изменение.

Чтобы экземпляр сущности перешел из временного состояния в хранимое, т. е. чтобы он стал управляемым, требуется либо вызвать метод `EntityManager#persist()`, либо создать ссылку из хранимого экземпляра и настроить каскадную передачу состояния для этой отображаемой связи.

Хранимое состояние

Экземпляр *хранимой* сущности имеет представление в базе данных. Он или уже хранится в базе данных, или будет сохранен по окончании выполнения единицы работы. Это экземпляр, обладающий идентичностью в базе данных, как показано в разделе 4.2; его идентификатору присвоено значение первичного ключа представления в базе данных.

Приложение может создавать экземпляры и делать их хранимыми, вызывая метод `EntityManager#persist()`. Также могут существовать экземпляры, ставшие хранимыми в результате создания ссылок на них из других, хранимых экземпля-

ров, которые уже управляются реализацией JPA. Экземпляр хранимой сущности может представлять объект, извлеченный из базы данных с помощью запроса, а также в результате поиска по идентификатору или навигации по графу объектов, начиная с другого хранимого экземпляра.

Хранимые экземпляры всегда связаны с контекстом хранения. Совсем скоро вы узнаете об этом больше.

Удаленное состояние

Удалить экземпляр хранимой сущности из базы даных можно несколькими способами. Например, вызовом метода `EntityManager#remove()`. Экземпляр можно также удалить, удалив ссылку на него из отображаемой коллекции с настроенным *удалением осиротевших объектов*.

В результате подобных действий экземпляр сущности оказывается в *удаленном состоянии*: реализация удалит его в конце выполнения единицы работы. Вы должны избавиться от всех ссылок на такой экземпляр после завершения работы с ним, например после вывода на экран сообщения, подтверждающего удаление.

Отсоединенное состояние

Для понимания идеи *отсоединенных* экземпляров сущностей рассмотрим процесс загрузки экземпляра. Чтобы получить экземпляр сущности по его (известному) идентификатору, вы вызываете метод `EntityManager#find()`. После этого вы завершаете выполнение единицы работы и закрываете контекст хранения. В приложении при этом остается *дескриптор* – ссылка на загруженный экземпляр. Он теперь находится в отсоединенном состоянии, и его данные перестают быть актуальными. Вы можете избавиться от ссылки, позволив сборщику мусора освободить занимаемую экземпляром память, или продолжить работу с данными в отсоединенном состоянии, а позже вызвать метод `merge()` для сохранения изменений в рамках новой единицы работы. Мы снова вернемся к теме отсоединения и слияния позже в этой главе, в специальном разделе.

У вас уже должно сложиться представление о состояниях экземпляров сущностей и переходах между ними. Нашей следующей темой станет контекст хранения – ключевая служба для любой реализации Java Persistence.

10.1.2. Контекст хранения

В приложении, использующем Java Persistence, объект `EntityManager` обладает контекстом хранения. Этот контекст создается вызовом метода `EntityManagerFactory#createEntityManager()`. Закрытие контекста осуществляется при помощи метода `EntityManager#close()`. В терминологии JPA это называется контекстом хранения, *управляемым приложением*; приложение определяет область действия контекста хранения, задавая рамки единицы работы.

Контекст хранения следит за всеми сущностями, находящимися в хранимом состоянии, и управляет ими. Контекст хранения – это центральная часть практически всех функций, предоставляемых реализацией JPA.

Контекст хранения позволяет механизму хранения выполнять *автоматическую проверку состояний объектов* (dirty checking), чтобы выявить экземпляры сущностей, модифицированные приложением. Затем реализация механизма хранения автоматически или по требованию синхронизирует с базой данных состояние экземпляров, находящихся под наблюдением контекста хранения. Обычно по завершении единицы работы реализация передает состояние из памяти в базу данных, выполняя SQL-инструкции `INSERT`, `UPDATE` и `DELETE`, которые являются частью языка управления данными DML (Data Modification Language). Процедура *выталкивания контекста* может выполняться и в другое время. Например, Hibernate может выполнять синхронизацию с базой данных перед выполнением запроса. Это гарантирует получение запросами изменений, произведенных ранее в рамках единицы работы.

Контекст хранения выступает в роли *кэша первого уровня*; он запоминает все экземпляры, вовлеченные в конкретную единицу работы. Например, если предложить Hibernate загрузить экземпляр сущности, используя значение первичного ключа (поиск по идентификатору), фреймворк может сначала проверить текущую единицу работы в контексте хранения. Если экземпляр сущности обнаружится в контексте хранения, никакого обращения к базе не будет – для приложения это будет повторным чтением. Последовательные вызовы метода `em.find(Item.class, ITEM_ID)` для одного контекста хранения будут возвращать одинаковый результат.

Этот кэш также влияет на результаты произвольных запросов, например выполняемых при помощи `javax.persistence.Query`. Hibernate читает результат SQL-запроса и превращает его в набор экземпляров сущностей. Сначала он пытается отыскать каждый экземпляр сущности в контексте хранения, используя поиск по идентификатору. И только если экземпляр с нужным значением идентификатора не найден в текущем контексте, Hibernate прочитает оставшуюся часть данных из набора, полученного в результате запроса. Если экземпляр сущности уже находится в контексте хранения, Hibernate проигнорирует все потенциально более новые данные из запроса из-за уровня изоляции транзакций в базе данных, при котором возможно чтение только подтвержденных данных (read commited).

Кэш контекста хранения работает всегда – его нельзя отключить. Он гарантирует следующее:

- уровень хранения не подвержен переполнению стека при наличии циклических ссылок в графе объектов;
- по окончании выполнения единицы работы не может существовать нескольких конфликтующих представлений одной и той же записи в базе данных. Реализация механизма хранения может спокойно записывать в базу данных любые изменения в экземплярах сущностей;
- аналогично все изменения в конкретном контексте хранения сразу становятся доступными всему коду, выполняемому внутри этой единицы работы и ее контекста хранения. JPA гарантирует целостность экземпляров сущностей.

Контекст хранения имеет *область гарантированной идентичности объектов*; в области действия контекста хранения строку базы данных может представлять лишь один экземпляр. Рассмотрим сравнение ссылок `entityA == entityB`. Это выражение вернет `true`, только если обе ссылки будут указывать на один и тот же Java-объект в куче. Теперь рассмотрим сравнение `entityA.getId().equals(entityB.getId())`. Это выражение вернет `true`, если оба объекта будут иметь одинаковое значение идентификатора в базе данных. Внутри одного контекста хранения Hibernate гарантирует, что оба сравнения дадут одинаковый результат. Это решает одну из фундаментальных проблем объектно-реляционного отображения, представленную в разделе 1.2.3.

Не лучше ли использовать идентичность, ограниченную областью видимости процесса?

Для типичного веб- или корпоративного приложения предпочтительнее использовать идентичность, ограниченную областью видимости контекста хранения. Идентичность, ограниченная областью видимости процесса, когда запись в базе данных представляет только один экземпляр в памяти процесса (JVM), дает некоторые потенциальные преимущества, касающиеся использования кэша. Однако в многопоточном приложении стоимость синхронизации доступа к хранимым экземплярам в глобальном словаре идентичностей будет слишком высокой. Более простое и масштабируемое решение состоит в том, чтобы каждый поток работал с отдельной копией данных в каждом контексте хранения.

Жизненный цикл экземпляров сущностей и служб, предоставляемых контекстом хранения, – сложная тема. Поэтому давайте рассмотрим несколько примеров проверки состояния объектов, кэширования и как область гарантированной идентичности работает на практике. Для этого нам понадобится диспетчер хранения `EntityManager`.

10.2. Интерфейс EntityManager

Любой инструмент поддержки прозрачности хранения включает диспетчера хранения. Этот диспетчер обычно предоставляет возможность выполнения базовых операций CRUD (создание, чтение, изменение, удаление), запросов и управления контекстом хранения. В приложении, использующем Java Persistence, основным интерфейсом для создания единиц работы является `EntityManager`.

10.2.1. Каноническая форма единицы работы

В Java SE и некоторых EE-архитектурах (например, если вы используете обычные сервлеты) получить экземпляр `EntityManager` можно вызовом метода `EntityManagerFactory#createEntityManager()`. Код приложения использует единственный экземпляр `EntityManagerFactory`, представляющий одну единицу хранения или

одну логическую базу данных. Большинство приложений использует лишь один общий экземпляр `EntityManagerFactory`.

Объект `EntityManager` применяется для выполнения одной единицы работы в одном потоке, и его создание является довольно дорогостоящей операцией. Листинг 10.1 демонстрирует типичную, каноническую форму единицы работы.

Листинг 10.1 ❖ Типичная форма единицы работы

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
EntityManager em = null;
UserTransaction tx = TM.getUserTransaction();
try {
    tx.begin();
    em = JPA.createEntityManager();  ← Управляется приложением
    // ...

    tx.commit();  ← Синхронизирует/выталкивает контекст хранения
} catch (Exception ex) {
    // Откат транзакции, обработка исключения
    // ...
} finally {
    if (em != null && em.isOpen())
        em.close();  ← Вы его создали – вы и закрывайте!
}
```

(Класс `TM` является вспомогательным классом и входит в состав примеров кода для этой книги. Здесь он нужен для упрощения поиска стандартного интерфейса `UserTransaction` в контексте JNDI. Класс `JPA` предоставляет удобный способ обращения к общему экземпляру `EntityManagerFactory`.)

Весь код между вызовами `tx.begin()` и `tx.commit()` выполняется в рамках одной транзакции. Пока что запомните, что все операции с базой данных в области действия транзакции, такие как выполняемые Hibernate SQL-запросы, либо выполняются успешно целиком, либо откатываются. Не думайте пока слишком много о транзакционном коде; вы прочтете больше о многопоточности в следующей главе. Мы рассмотрим тот же пример снова, обращая внимание на транзакции и код обработки ошибок. Не пишите пустых блоков `catch` в своем коде – вы должны откатывать транзакции и обрабатывать исключения.

Создание экземпляра `EntityManager` запускает его контекст хранения. Hibernate не будет обращаться к базе данных без необходимости; экземпляр `EntityManager` не извлекает соединения JDBC `Connection` из пула, пока не потребуется выполнить SQL-запрос. Вы можете создать и закрыть объект `EntityManager`, так и не выполнив ни одной операции с базой данных. Hibernate выполняет выражения SQL, когда вы ищите или запрашиваете данные или когда он записывает изменения, обнаруженные в контексте хранения. В процессе создания объекта `EntityManager` фреймворк Hibernate присоединяется к текущей системной транзакции и ожидает ее подтверждения. Когда Hibernate получит уведомление (от механизма JTA) о подтверждении транзакции, он выполнит проверку состояния объектов в кон-

тексте хранения и произведет синхронизацию с базой данных. Проверку состояния объектов и синхронизацию можно выполнить принудительно в любое время в течение транзакции, с помощью метода `EntityManager#flush()`.

Вы можете определять границы контекста хранения, выбирая момент вызова метода `close()` объекта `EntityManager`. В какой-то момент вам придется закрыть контекст, поэтому всегда помещайте вызов `close()` в блок `finally`.

Как долго контекст хранения может находиться открытым? Для последующих примеров предположим, что мы пишем сервер и каждый клиентский запрос будет обрабатываться в отдельном контексте хранения и в рамках своей транзакции. Если вы знакомы с сервлетами, представьте, что код в листинге 10.1 находится внутри метода `service()`. В рамках этой единицы работы вы получаете доступ к экземпляру `EntityManager` для загрузки и сохранения данных.

10.2.2. Сохранение данных

Давайте создадим новый экземпляр сущности и переведем его в хранимое состояние:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = new Item();
item.setName("Some Item");  ← Для поля Item#name определено ограничение NOT NULL

em.persist(item);

Long ITEM_ID = item.getId();  ← Уже присвоен
```

Эта единица работы и изменение состояния экземпляра `Item` изображены на рис. 10.2.

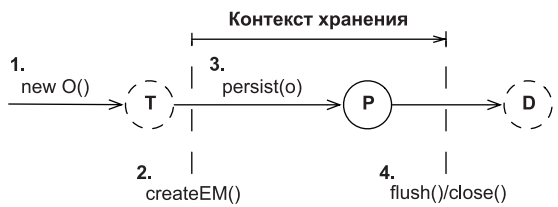


Рис. 10.2 ❖ Переход экземпляра в хранимое состояние в рамках единицы работы

Новый временный экземпляр `Item` создается как обычно. Конечно, его можно было бы создать перед объектом `EntityManager`. Вызов метода `persist()` переводит временный экземпляр `Item` в хранимое состояние. Теперь он стал управляемым и связан с текущим контекстом хранения.

Для записи экземпляра `Item` в базу данных Hibernate должен выполнить SQL-инструкцию `INSERT`. После подтверждения транзакция этой единицы работы Hibernate вытолкнет контекст хранения, и в этот момент выполнится инструкция

INSERT. Hibernate может выполнять сразу несколько инструкций INSERT на уровне JDBC. Вызов метода persist() лишь присвоит значение идентификатора объекту Item. Иначе, если генератор идентификаторов не вызывается *перед вставкой*, выражение INSERT будет выполнено в момент вызова persist(). Возможно, вы захотите заново просмотреть раздел 4.2.5.

Определение состояния сущности при помощи идентификатора

Иногда требуется узнать, является ли экземпляр сущности временным, хранимым или отсоединенным. Экземпляр находится в хранимом состоянии, если вызов EntityManager#contains(e) возвращает true; во временном, если вызов PersistenceUnitUtil#getIdentifier(e) возвращает null, и в отсоединенном, если не является хранимым и метод PersistenceUnitUtil#getIdentifier(e) возвращает значение поля идентификатора сущности. Получить доступ к PersistenceUnitUtil можно из объекта EntityManagerFactory.

Обратите внимание на два момента. Во-первых, значение идентификатора может быть не присвоено и недоступно до момента выталкивания контекста хранения. Во-вторых, Hibernate (в отличие от некоторых других реализаций JPA) никогда не вернет значения null из метода PersistenceUnitUtil#getIdentifier(), если поле идентификатора имеет простой тип (например, long, а не Long).

Желательно (но необязательно) полностью инициализировать экземпляр Item перед включением в контекст хранения. SQL-выражение INSERT содержит значения, которые находились в экземпляре в момент вызова persist(). Если не присвоить значение полю name объекта Item, перед тем как сделать его хранимым, может быть нарушено ограничение NOT NULL. Изменения в объекте Item, произведенные после вызова persist(), попадут в базу данных в результате выполнения дополнительного SQL-выражения UPDATE.

Если одно из выражений – INSERT или UPDATE – приведет к ошибке во время сохранения контекста, Hibernate отменит изменения, сделанные в рамках данной транзакции, на уровне базы данных, но не отменит изменений в памяти. Если поменять значение поля Item#name после вызова persist(), ошибка при подтверждении транзакции не вернет прежнего значения. Это вполне логично, поскольку восстановиться после ошибки в транзакции обычно невозможно, и вы должны немедленно избавиться от поврежденного контекста хранения и объекта EntityManager. Мы обсудим обработку исключений в следующей главе.

Далее мы будем загружать и модифицировать сохраненные данные.

10.2.3. Извлечение и модификация хранимых данных

Извлекать хранимые экземпляры из базы данных можно с помощью объекта EntityManager. В следующем примере предполагается, что где-то имеется значение идентификатора объекта Item, сохраненного в предыдущем разделе, и теперь нужно найти этот объект по идентификатору в рамках новой единицы работы:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = em.find(Item.class, ITEM_ID);
if (item != null)
    item.setName("New Name");
```

Обратиться к базе данных, если объект отсутствует в контексте хранения

Изменение состояния

Рисунок 10.3 наглядно демонстрирует этот переход.

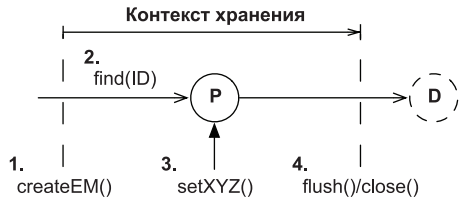


Рис. 10.3 ❖ Переход экземпляра в хранимое состояние в рамках единицы работы

Значение, возвращаемое методом `find()`, не требует приведения типа; это параметризованный метод, тип результата которого определяется первым параметром. Возвращаемый экземпляр сущности находится в хранимом состоянии, и его можно изменить в рамках единицы работы.

Если в контексте нет ни одного хранимого экземпляра с заданным идентификатором, метод `find()` вернет `null`. Операция `find()` всегда обращается к базе данных, если сущность требуемого типа с искомым идентификатором отсутствует в кэше контекста хранения. Экземпляр сущности всегда инициализируется во время загрузки. Вы можете быть уверены в том, что все его значения будут доступны позже в отсоединенном состоянии: например, при отображении на экране после закрытия контекста хранения. (Hibernate может не обращаться к базе данных, если включен необязательный кэш второго уровня; мы рассмотрим этот кэш в разделе 20.2.)

Вы можете изменить экземпляр `Item`, а контекст хранения автоматически обнаружит эти изменения и запишет их в базу данных. Когда Hibernate выталкивает контекст хранения во время подтверждения транзакции, он выполняет необходимые выражения SQL DML для синхронизации изменений с базой данных. Hibernate старается передать изменения в базу данных как можно позже после завершения транзакции. Инструкции DML обычно устанавливают в базе данных блокировку, существующую до конца транзакции, так что Hibernate сводит продолжительность блокировки к минимуму.

Hibernate запишет новое значение поля `Item#name` в базу данных, выполнив SQL-выражение `UPDATE`. По умолчанию Hibernate включает в это выражение все столбцы отображаемой таблицы `ITEM`, перезаписывая старые значения заново в неизмененных столбцах. Если понадобится включать в SQL-инструкции только измененные столбцы (или, в случае с `INSERT`, только столбцы, которые не могут содержать null), можно настроить динамическую генерацию кода SQL, как было показано в разделе 4.3.2.

Hibernate обнаружит изменение в поле `name`, сравнивая объект `Item` с копией, сделанной ранее, в момент загрузки объекта `Item` из базы данных. Если объект отличается от копии, значит, требуется выполнить операцию `UPDATE`. Копия состояния в контексте хранения расходует память. Проверка состояния объектов с использованием копий также увеличивает накладные расходы, поскольку во время выталкивания контекста Hibernate должен сравнить все экземпляры в контексте с их копиями.

Вам может понадобиться настроить порядок выявления изменений, используя механизм расширения. Укажите в параметре `hibernate.entity_dirtiness_strategy` (в файле конфигурации persistence.xml) имя класса, реализующего интерфейс `org.hibernate.CustomEntityDirtinessStrategy`. Чтобы узнать больше об этом интерфейсе, прочтите его документацию Javadoc. Интерфейс `org.hibernate.Interceptor` представляет еще один механизм управления проверкой состояния объектов путем реализации его метода `findDirty()`. Пример реализации этого обработчика можно найти в разделе 13.2.2.

Ранее упоминалось, что контекст хранения позволяет повторно читать экземпляры сущностей и предоставляет гарантию объектной идентичности:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item itemA = em.find(Item.class, ITEM_ID);
Item itemB = em.find(Item.class, ITEM_ID);  ← Повторное чтение

assertTrue(itemA == itemB);
assertTrue(itemA.equals(itemB));
assertTrue(itemA.getId().equals(itemB.getId()));
```

Первый вызов `find()` обращается к базе данных и получает объект `Item` с помощью выражения `SELECT`. Второй находит искомый экземпляр в контексте хранения и возвращает кэшированный объект `Item`.

Иногда может потребоваться получить экземпляр сущности без обращения к базе данных.

10.2.4. Получение ссылки на объект

Если вы не хотите обращаться к базе данных во время загрузки экземпляра сущности, поскольку не знаете, понадобится ли вам полностью инициализированный экземпляр, можно попросить объект `EntityManager` загрузить пустой указатель – прокси-объект:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = em.getReference(Item.class, ITEM_ID);  ← ❶ getReference()

PersistenceUnitUtil persistenceUtil =  ← ❷ Вспомогательный метод
    JPA.getEntityManagerFactory().getPersistenceUnitUtil();
assertFalse(persistenceUtil.isLoaded(item));

// assertEquals(item.getName(), "Some Item");  ← ❸ Инициализация прокси-объекта
// Hibernate.initialize(item);  ← ❹ Загрузка данных из прокси-объекта
```

```
tx.commit();
em.close();

assertEquals(item.getName(), "Some Item");   ← ❺ Объект находится в отсоединенном состоянии
```

❶ Если контекст хранения уже содержит объект `Item` с заданным идентификатором, он будет возвращен методом `getReference()` без обращения к базе данных. Если в данный момент в управляемом состоянии нет *ни одного* хранимого экземпляра с таким идентификатором, Hibernate создаст прокси-объект. Это значит, что метод `getReference()` не будет обращаться к базе данных, но и не вернет `null`, в отличие от метода `find()`.

❷ JPA предоставляет в классе `PersistenceUnitUtil` вспомогательный метод `isLoaded()`, помогающий определить факт работы с неинициализированным прокси-объектом.

❸ Стоит только вызвать какой-нибудь метод прокси-объекта, например `Item#getName()`, как тут же будет выполнена инструкция `SELECT` для его полной инициализации. Исключением из правила является метод `getId()` получения отображаемого идентификатора из базы данных. Прокси-объект может выглядеть как настоящий объект, но это всего лишь пустышка, в которой хранится значение идентификатора представляемого им экземпляра сущности. Если во время инициализации прокси-объекта соответствующей записи в базе данных не окажется, будет возбуждено исключение `EntityNotFoundException`. Обратите внимание, что это исключение может быть возбуждено также при вызове метода `Item#getName()`.

❹ В Hibernate имеется удобный статический метод `initialize()` для загрузки данных прокси-объекта.

❺ После закрытия контекста хранения объект `Item` оказывается в отсоединенном состоянии. Если прокси-объект не был инициализирован, пока контекст хранения был открыт, при обращении к прокси-объекту будет возбуждено исключение `LazyInitializationException`. Нельзя загружать данные по требованию после закрытия контекста. Но эта проблема имеет простое решение – загружайте данные перед закрытием контекста хранения.

Мы еще многое вам расскажем о прокси-объектах, отложенной загрузке и извлечении данных по требованию в главе 12.

Чтобы удалить экземпляр сущности из базы данных, его нужно перевести во временное состояние.

10.2.5. Переход данных во временное состояние

Чтобы перевести экземпляр сущности во временное состояние и удалить его из базы данных, вызовите метод `remove()` объекта `EntityManager`:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = em.find(Item.class, ITEM_ID);   ← ❶ Вызвать find() или getReference()
// Item item = em.getReference(Item.class, ITEM_ID);

em.remove(item);   ← ❷ Поместить экземпляр в очередь на удаление

assertFalse(em.contains(item));   ← ❸ Проверка состояния сущности

// em.persist(item);   ← ❹ Отмена удаления

assertNull(item.getId());   ←  Был включен параметр hibernate.use_identifier_rollback;
                               теперь этот экземпляр выглядит как временный
tx.commit();
em.close();
```

❶ Если вызвать метод `find()`, Hibernate выполнит выражение `SELECT` и загрузит объект `Item`. Если вызвать метод `getReference()`, Hibernate попытается обойтись без инструкции `SELECT` и вернет прокси-объект.
❷ Вызов метода `remove()` добавит экземпляр сущности в очередь на удаление по окончании выполнения единицы работы; теперь он находится в *удаленном* состоянии. Если метод `remove()` вызвать для прокси-объекта, Hibernate выполнит инструкцию `SELECT`, чтобы загрузить данные. Экземпляр сущности должен быть полностью инициализирован перед переходом между состояниями жизненного цикла. У вас могут быть определены методы обратных вызовов для событий жизненного цикла или настроены обработчики (см. раздел 13.2), и экземпляр должен быть обработан каждым из них для завершения своего жизненного цикла.
❸ Сущность в удаленном состоянии больше не является хранимой. Это можно проверить вызовом `contains()`.
❹ Отменив уделение, экземпляр можно снова сделать хранимым.
❺ После подтверждения транзакции Hibernate синхронизирует переходы между состояниями с базой данных и выполнит SQL-выражение `DELETE`. Сборщик мусора JVM обнаружит отсутствие ссылок на `item` и удалит последние остатки данных из памяти.

Этот процесс показан на рис. 10.4.

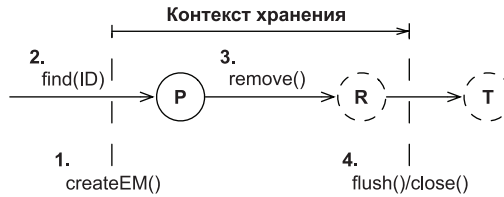


Рис. 10.4 ❖ Удаление экземпляра в рамках единицы работы

По умолчанию Hibernate не меняет значения идентификатора удаляемого экземпляра сущности. Это значит, что `item.getId()` по-прежнему будет возвращать устаревшее значение идентификатора. Иногда бывает полезно продолжить работать с «удаленными» данными и дальше – например, удаленный объект `Item` можно сохранить на случай, если пользователь захочет отменить операцию. Как показано в примере, можно вызвать метод `persist()` для удаленного экземпляра и отменить удаление перед выталкиванием контекста. С другой стороны, если присвоить параметру `hibernate.use_identifier_rollback` значение `true` в файле конфигурации persistence.xml, Hibernate сбросит значение идентификатора после удаления экземпляра сущности. В предыдущем примере идентификатору было присвоено значение по умолчанию `null` (поскольку он имеет тип `Long`). Теперь объект `Item` оказывается во временном состоянии, и его можно снова сохранить в новом контексте хранения.

Java Persistence также предоставляет массовые операции, транслирующиеся непосредственно в SQL-выражения `DELETE`, не вовлекающие в работу обработчиков событий жизненного цикла. Мы обсудим эти операции в разделе 20.1.

Предположим, что вы загрузили экземпляр сущности из базы данных и работаете с его данными. Каким-то образом вам стало известно, что другое приложение или другой поток вашего приложения изменил соответствующую запись в базе данных. Давайте теперь посмотрим, как обновить данные, загруженные в память.

10.2.6. Изменение данных в памяти

Следующий пример демонстрирует изменение экземпляра хранимой сущности:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
Item item = em.find(Item.class, ITEM_ID);
item.setName("Some Name");

// Кто-то обновил эту запись в базе данных

String oldName = item.getName();
em.refresh(item);
assertNotEquals(item.getName(), oldName);
```

После загрузки экземпляра сущности вы поняли (не важно, как), что кто-то изменил запись в базе данных. После вызова `refresh()` Hibernate выполнит выражение `SELECT`, чтобы прочитать и загрузить результаты запроса, замещая все изменения, произведенные в хранимой сущности в памяти. Если запись в базе данных больше не существует (кто-то ее удалил), при вызове `refresh()` Hibernate возбудит исключение `EntityNotFoundException`.

Большинству приложений не требуется обновлять состояние в памяти вручную; конфликты, вызванные изменениями в многопоточной среде, обычно разрешаются во время подтверждения транзакции. А операцию обновления лучше использовать с расширенным контекстом хранения, который может оставаться открытым на протяжении нескольких циклов запрос/ответ и/или системных транзакций. Пока вы ожидаете ввода данных пользователем с открытым контекстом хранения, данные могут устареть, и может понадобиться выборочное обновление, в зависимости от длительности диалогового взаимодействия между пользователем и системой. Обновление может оказаться полезным для отмены изменений, сделанных в памяти в процессе диалогового взаимодействия, например если пользователь его прервет. Мы еще вернемся к теме обновления в процессе диалогового взаимодействия с пользователем в разделе 18.3.

Другой нечасто используемой операцией является репликация экземпляра сущности.

10.2.7. Репликация данных

Репликация полезна, когда нужно извлечь данные из одной базы данных и сохранить в другую. Процесс репликации получает отсоединенные экземпляры, загруженные в один контекст хранения, и сохраняет их в другом контексте. Обычно контексты открываются с помощью двух конфигураций `EntityManagerFactory`, описывающих две логические базы данных. Вы должны отобразить сущность в каждой из конфигураций.

Операция `replicate()` доступна только в Hibernate-интерфейсе `Session`. Следующий пример загружает экземпляр `Item` из одной базы данных и копирует в другую:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
tx.begin();

EntityManager emA = getDatabaseA().createEntityManager();
Item item = emA.find(Item.class, ITEM_ID);

EntityManager emB = getDatabaseB().createEntityManager();
emB.unwrap(Session.class)
    .replicate(item, org.hibernate.ReplicationMode.LATEST_VERSION);

tx.commit();
emA.close();
emB.close();
```

Соединения с обеими базами данных могут участвовать в одной системной транзакции.

Выполнением процедуры репликации управляет перечисление `ReplicationMode`:

- `IGNORE`. Игнорировать экземпляр, если в базе данных уже есть запись с таким же идентификатором;
- `OVERWRITE`. Перезаписать существующую запись с таким же идентификатором;
- `EXCEPTION`. Возбудить исключение, если в целевой базе данных найдется запись с таким же идентификатором;
- `LATEST_VERSION`. Перезаписать существующую запись, если ее версия старше версии данного экземпляра сущности, иначе игнорировать экземпляр. Требует использования оптимистического управления конкурентным доступом с версионированием сущностей (см. раздел 11.2.2).

Репликация может понадобиться, например, для приведения в соответствие данных в различных базах данных. Типичным примером может служить обновление программы: если новая версия приложения требует новой (схемы) базы данных, вам могут понадобиться одновременная миграция и репликация данных.

Контекст хранения многое сделает за вас: автоматическую проверку состояния объектов, обеспечение гарантированной области идентичности объектов и т. д. Но также важно, чтобы вы знали некоторые подробности управления им и могли иногда влиять на происходящее за кулисами.

10.2.8. Кэширование в контексте хранения

Контекст хранения служит кэшем хранимых экземпляров. Каждый экземпляр сущности в хранимом состоянии связан с контекстом хранения.

Большинство пользователей Hibernate, игнорирующих этот простой факт, получает исключение `OutOfMemoryException`. Обычно это происходит при попытке загрузить тысячи экземпляров сущностей в рамках единицы работы, которые никог-

да не меняются. Для каждого экземпляра Hibernate должен сделать копию в кэше контекста хранения, что может привести к исчерпанию памяти (очевидно, что для изменения тысяч записей нужно использовать массовые операции; мы вернемся к подобному типу единицы работы в разделе 20.1).

Кэш контекста хранения никогда не уменьшается автоматически. Поэтому удерживайте его размер на минимально необходимом уровне. Как правило, большинство хранимых экземпляров оказывается в контексте случайно, например если требовалось несколько объектов, а вы запросили больше. Чрезмерно большие графы могут сильно влиять на производительность и требовать значительного расхода памяти для хранения копий. Убедитесь, что ваши запросы возвращают только необходимые данные, и обратите внимание на следующие способы управления поведением кэша Hibernate.

Вы можете вызвать метод `EntityManager#detach(i)` для удаления хранимого экземпляра из контекста вручную. Также можно вызвать `EntityManager#clear()` для отсоединения всех хранимых экземпляров, оставляя контекст хранения пустым.

Оригинальный интерфейс `Session` предоставляет несколько полезных операций. Например, весь контекст хранения можно перевести в режим доступа только на чтение. Это предотвратит создание копий и отключит проверку состояния объектов, и Hibernate не будет записывать изменений в базу данных:

Файл: /examples/src/test/java/org/jpwh/test/fetching/ReadOnly.java

```
em.unwrap(Session.class).setDefaultReadOnly(true);

Item item = em.find(Item.class, ITEM_ID);
item.setName("New Name");

em.flush();  ← Никакой операции UPDATE
```

Можно отключить проверку состояния для единственного экземпляра сущности:

Файл: /examples/src/test/java/org/jpwh/test/fetching/ReadOnly.java

```
Item item = em.find(Item.class, ITEM_ID);

em.unwrap(Session.class).setReadOnly(item, true);

item.setName("New Name");

em.flush();  ← Никакой операции UPDATE
```

Запрос, выполненный интерфейсом `org.hibernate.Query`, может возвращать результаты, доступные только для чтения, изменения которых не будут проверяться:

Файл: /examples/src/test/java/org/jpwh/test/fetching/ReadOnly.java

```
org.hibernate.Query query = em.unwrap(Session.class)
    .createQuery("select i from Item i");

query.setReadOnly(true).list();

List<Item> result = query.list();
```

```
for (Item item : result)
    item.setName("New Name");

em.flush();  ← Никакой операции UPDATE
```

С помощью подсказок в запросах можно также отключить проверку состояния объектов для экземпляров, полученных с помощью стандартного JPA-интерфейса `javax.persistence.Query`:

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.READ_ONLY,
        true
    );
```

Будьте осторожны при работе с экземплярами сущностей, доступными только для чтения, – их по-прежнему можно удалять, а модификации коллекций могут преподносить сюрпризы! Руководство Hibernate содержит длинный список особых случаев, который нужно прочитать перед использованием этих настроек с отображаемыми коллекциями. Вы увидите больше примеров запросов в главе 14.

До сих пор выталкивание контекста хранения и синхронизация происходили автоматически, во время подтверждения транзакции. В некоторых ситуациях может понадобиться получить более полный контроль над процессом синхронизации.

10.2.9. Выталкивание контекста хранения

По умолчанию Hibernate выталкивает контекст хранения объекта `EntityManager` и синхронизирует изменения с базой данных, как только происходит подтверждение присоединенной транзакции. Эту стратегию использовали все предыдущие примеры, кроме некоторых в предыдущем разделе. JPA позволяет реализациям выполнять синхронизацию контекста хранения в другие моменты, если они того желают.

Hibernate как реализация JPA выполняет синхронизацию:

- во время подтверждения присоединенной системной транзакции JTA;
- перед выполнением запроса; мы не имеем в виду поиск с помощью `find()`, а запрос, выполненный с помощью интерфейса `javax.persistence.Query` или другого подобного Hibernate API;
- во время явного вызова `flush()` приложением.

Это поведение нельзя изменить с помощью настройки `FlushModeType` объекта `EntityManager`:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);  ← ❶ Загрузка экземпляра Item
item.setName("New Name");  ← ❷ Изменение имени экземпляра
```

```
em.setFlushMode(FlushModeType.COMMIT);
assertEquals(
    em.createQuery("select i.name from Item i where i.id = :id")
        .setParameter("id", ITEM_ID).getSingleResult(),
    "Original Name"
);

tx.commit();
em.close();
```

Отключение сохранения контекста перед выполнением запроса

❸ Получение имени экземпляра

Выталкивание контекста!

Этот пример загружает товар (Item) ❶ и меняет его имя ❷. После этого выполняется запрос к базе данных и извлекается имя товара ❸. Обычно Hibernate определяет факт изменения данных в памяти и синхронизирует изменения с базой данных перед выполнением запроса. Такое поведение соответствует значению `FlushModeType.AUTO`, которое используется по умолчанию во время присоединения объекта `EntityManager` к транзакции. Используя значение `FlushModeType.COMMIT`, можно отключить выталкивание контекста перед выполнением запроса и, следовательно, наблюдать различия между данными в памяти и полученными в результате запроса. Синхронизация произойдет только во время подтверждения транзакции.

Вызывая `EntityManager#flush()`, пока выполняется транзакция, можно в любой момент инициировать проверку состояния объектов и синхронизацию с базой данных.

На этом мы завершим обсуждение *временного*, *хранимого* и *удаленного* состояний сущностей, а также основных применений интерфейса `EntityManager`. Понимание переходов между состояниями и методов API очень важно; ни одно JPA-приложение не обходится без этих операций.

Далее мы рассмотрим *отсоединенное* состояние сущности. Мы уже касались некоторых проблем, возникающих при удалении связи между экземплярами сущностей с контекстом хранения, таких как отключение отложенной инициализации. Давайте рассмотрим отсоединенное состояние на нескольких примерах, чтобы вы знали, чего ожидать во время работы с данными вне контекста хранения.

10.3. Работа с отсоединенным состоянием

Ссылку, покинувшую область гарантированной идентичности, мы называем *ссылкой* на *экземпляр отсоединенной сущности*. После закрытия контекст хранения больше не обеспечивает отображения идентичности. При работе с отсоединенными экземплярами сущностей вы столкнетесь с проблемой псевдонимов (aliasing), поэтому вы должны понимать, как работать с идентичностями отсоединенных экземпляров.

10.3.1. Идентичность отсоединенных экземпляров

Если выполнить поиск данных, используя один и тот же идентификатор в базе данных в одном контексте хранения, в результате получатся две ссылки на один объект в памяти JVM. Далее показано выполнение двух единиц работы.

Листинг 10.2 ❖ Области гарантированной идентичности объектов в Java Persistence

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
tx.begin();  ← ❶ Создание контекста хранения
em = JPA.createEntityManager();  ← ❷ Загрузка экземпляров сущностей

Item a = em.find(Item.class, ITEM_ID);
Item b = em.find(Item.class, ITEM_ID);
assertTrue(a == b);        ← ❸ Объекты a и b идентичны
assertTrue(a.equals(b));  ← ❹ Ссылки a и b равны
assertEquals(a.getId(), b.getId());  ← ❺ a и b обладают одной идентичностью в базе данных

tx.commit();  ← ❻ Подтверждение транзакции
em.close();  ← ❼ Закрытие контекста хранения

tx.begin();
em = JPA.createEntityManager();

Item c = em.find(Item.class, ITEM_ID);
assertTrue(a != c);  ← ❽ Ссылки a и c не идентичны
assertFalse(a.equals(c));  ← ❾ Ссылки a и c не равны
assertEquals(a.getId(), c.getId());  ← ❿ Идентичность в базе данных остается истинной

tx.commit();
em.close();
```

В первой единице работы, начинающейся с вызова `begin()` ❶, создается контекст хранения ❷ и загружается несколько экземпляров сущностей. Поскольку ссылки `a` и `b` получены из одного контекста, в Java они будут идентичными ❸. Они также будут равны ❹, так как по умолчнию метод `equals()` полагается на сравнение идентичностей в Java. Очевидно, что у них одна идентичность в базе данных ❺. Они ссылаются на один хранимый экземпляр `Item`, управляемый контекстом хранения в рамках этой единицы работы. Первая часть этого примера завершается подтверждением транзакции ❻ и закрытием контекста хранения ❼. После закрытия первого контекста ссылки `a` и `b` будут находиться в отсоединенном состоянии. Работа продолжится с экземплярами, существующими вне области гарантированной идентичности объектов.

Легко видеть, что ссылки `a` и `c`, загруженные в разных контекстах хранения, неидентичны ❽. Проверка на равенство с помощью `a.equals(c)` дает результат `false` ❾. Проверка на идентичность в базе данных по-прежнему возвращает `true` ❿. Такое поведение может создавать проблемы, если рассматривать экземпляры сущностей в отсоединенном состоянии как равные. Рассмотрим, к примеру, следующий фрагмент, добавленный после окончания второй единицы работы:

```
em.close();

Set<Item> allItems = new HashSet<>();
allItems.add(a);
allItems.add(b);
allItems.add(c);
assertEquals(allItems.size(), 2);  ← Выглядит как произвольный и неверный результат
```

В этом примере все три ссылки помещаются в коллекцию `Set`. Все они ссылаются на отсоединенные экземпляры. Если теперь проверить размер коллекции – количество элементов, какой будет результат?

Множество `Set` не позволяет хранить повторяющиеся элементы. Они обнаруживаются коллекцией `Set`; при попытке добавить ссылку тут же автоматически выполнится метод `Item#equals()` для сравнения со всеми элементами в коллекции. Если для какого-нибудь элемента `equals()` вернет `true`, добавления не произойдет.

По умолчанию все классы Java наследуют реализацию метода `equals()` от `java.lang.Object`. Для проверки ссылок на экземпляры в Java-куче эта реализация использует оператор равенства (`==`).

Можно догадаться, что в коллекции окажутся два элемента. Все-таки `a` и `b` ссылаются на один экземпляр в памяти; они были загружены в один контекст хранения. Ссылка `c` получена из другого контекста; она указывает на другой экземпляр в куче. Всего в примере имеются три ссылки на два экземпляра, но мы знаем это лишь потому, что видели код загрузки данных. В реальном приложении может быть неизвестно, что `a` и `b` загружены в контекст, отличный от контекста `c`. Кроме того, кто-то из вас мог бы ожидать, что в коллекции окажется один элемент, поскольку `a`, `b` и `c` представляют одну запись в базе данных, один экземпляр `Item`.

Каждый раз, когда приходится работать с экземплярами в отсоединенном состоянии и проверять их равенство (обычно в коллекциях, основанных на значениях хэша), определяйте свою реализацию методов `equals()` и `hashCode()` для отображаемых классов сущностей. Это важный нюанс: если не приходится работать с экземплярами сущностей в отсоединенном состоянии, никаких действий предпринимать не нужно, и реализация `equals()` по умолчанию в классе `java.lang.Object` вполне подойдет. Вы полагаетесь на область гарантированной идентичности объектов внутри контекста хранения, предоставляемую Hibernate. Даже если вы работаете с отсоединенными экземплярами, но не проверяете их равенство, не помещаете в коллекцию `Set` и не используете как ключи словаря `Map`, вам не о чем беспокоиться. Если вы просто отображаете экземпляр `Item` на экране, он не будет ни с чем сравниваться.

Большинство разработчиков, только начинающих работу с JPA, считает, что они должны определить собственную реализацию проверки на равенство для каждого класса сущности, но это не так. В разделе 18.3 мы покажем проект приложения, использующий стратегию работы с *расширенным* контекстом хранения. Эта стратегия также расширяет область гарантированной идентичности объектов, распространяя ее на весь процесс диалогового взаимодействия с пользователем и несколько системных транзакций. Заметьте, что по-прежнему не следует сравнивать отсоединенные экземпляры, полученные в контексте разных диалоговых взаимодействий!

Предположим, что вы хотите использовать отсоединенные экземпляры и проверять их равенство, используя собственный метод.

10.3.2. Реализация метода проверки равенства

Реализовать методы `equals()` и `hashCode()` можно разными способами. Но помните, что, переопределяя `equals()`, обязательно следует переопределить `hashCode()`, чтобы они вели себя согласованно. Если два экземпляра равны, они должны иметь одинаковые значения хэша.

Кажется, что в `equals()` достаточно было бы сравнивать только свойства идентификаторов в базе данных, как правило, представляющих суррогатные первичные ключи. По сути, если два экземпляра `Item` имеют одинаковое значение идентификатора, возвращаемое методом `getId()`, они представляют одну сущность. Если метод `getId()` возвращает `null`, это временный экземпляр `Item`, который еще не был сохранен.

К сожалению, это решение имеет один большой недостаток – Hibernate не присваивает идентификаторов, пока экземпляры не станут хранимыми. Если перед сохранением добавить временный экземпляр в коллекцию `Set`, после сохранения его хэш изменится, пока он все еще будет находиться в коллекции `Set`. Это противоречит контракту коллекции `java.util.Set` и нарушит ее работу. В частности, из-за этой проблемы становится бесполезным использование каскадной передачи хранимого состояния в отображаемых связях на основе множеств. Мы выступаем решительно против использования равенства идентификаторов в базе данных.

Чтобы подобраться к решению, мы советуем познакомиться с понятием *бизнес-ключа*. Бизнес-ключ – это свойство или их комбинация, уникальное для каждого экземпляра с одинаковым идентификатором в базе данных. По сути, это естественный ключ, который можно было бы использовать взамен суррогатного первичного ключа. В отличие от естественного первичного ключа, к бизнес-ключу не предъявляется требование его неизменности – достаточно и того, чтобы он изменялся редко.

Мы утверждаем, что практически каждый класс сущности должен обладать бизнес-ключом, даже если он включает в себя все свойства этого класса (что вполне подходит для некоторых неизменяемых классов). По каким признакам ваши пользователи будут различать A, B и C в списке товаров на экране? Это свойство или их комбинация и будет вашим бизнес-ключом. Бизнес-ключ – это то, что по мнению пользователя уникально идентифицирует конкретную запись, тогда как суррогатный ключ – это то, на что полагаются приложение и СУБД. Наиболее вероятно, что свойство или свойства, составляющие бизнес-ключ, будут иметь в схеме базы данных ограничение `UNIQUE`.

Давайте создадим собственные методы проверки равенства для класса сущности `User`; это проще, чем сравнение экземпляров `Item`. В случае класса `User` на роль бизнес-ключа идеально подходит свойство `username`. Это обязательное поле, имеющее ограничение уникальности в базе данных, которое меняется редко (если меняется вообще).

Листинг 10.3 ❖ Собственный метод определения равенства объектов `User`

```
@Entity
@Table(name = "USERS",
       uniqueConstraints =
        @UniqueConstraint(columnNames = "USERNAME"))
```

```
public class User {

    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (!(other instanceof User)) return false;  ← Используйте оператор instanceof
        User that = (User) other;
        return
     this.getUsername().equals(that.getUsername());  ← Используйте методы чтения
    }

    @Override
    public int hashCode() {
        return getUsername().hashCode();
    }

    // ...
}
```

Вы, должно быть, заметили, что метод `equals()` всегда обращается к полям «другого» объекта с помощью методов чтения. Это чрезвычайно важно, поскольку ссылка `other` может оказаться прокси-объектом, а не обычным экземпляром, несущим свое хранимое состояние. Нельзя напрямую обратиться к полю `username` класса `User` через прокси-объект. Чтобы инициализировать прокси-объект для получения значения свойства, нужно обратиться к нему, используя метод чтения. Это одна из ситуаций, когда применение Hibernate не является *абсолютно* прозрачным, но тем не менее использование методов чтения вместо непосредственного обращения к переменным экземпляра является хорошей практикой.

Тип ссылки `other` следует проверять с помощью оператора `instanceof`, а не сравнением значений, возвращаемых методом `getClass()`. Опять же, ссылка `other` может оказаться прокси-объектом, представляющим экземпляр сгенерированного во время выполнения подкласса `User`, поэтому типы `this` и `other` могут оказаться разными, но при этом находиться в отношении супертип/подтип. Вы можете узнать больше о прокси-объектах в разделе 12.1.1.

Теперь вы можете без проблем сравнивать объекты `User`, находящиеся в хранимом состоянии:

```
tx.begin();
em = JPA.createEntityManager();

User a = em.find(User.class, USER_ID);
User b = em.find(User.class, USER_ID);
assertTrue(a == b);
assertTrue(a.equals(b));
assertEquals(a.getId(), b.getId());

tx.commit();
em.close();
```

Кроме того, вы получаете корректное поведение при сравнении ссылок на хранимые и отсоединенные экземпляры:

```
tx.begin();
em = JPA.createEntityManager();

User c = em.find(User.class, USER_ID);
assertFalse(a == c);       ← Значение, конечно же, все еще false
assertTrue(a.equals(c));   ← Теперь true
assertEquals(a.getId(), c.getId());

tx.commit();
em.close();

Set<User> allUsers = new HashSet();
allUsers.add(a);
allUsers.add(b);
allUsers.add(c);
assertEquals(allUsers.size(), 1);   ← Правильно!
```

Для других сущностей бизнес-ключ может быть более сложным и состоять из комбинации свойств. Вот несколько советов, которые помогут вам в выявлении бизнес-ключей классов предметной модели.

- Посмотрите, по каким атрибутам пользователи вашего приложения идентифицируют объекты (в реальном мире). Как пользователи отличают один элемент от другого, когда они отображаются на экране? Возможно, это и будет искомым бизнес-ключом.
- Каждый неизменяемый атрибут является потенциальным кандидатом на роль бизнес-ключа. Изменяемые атрибуты тоже могут быть хорошими кандидатами, если они редко изменяются или если их изменение находится под вашим контролем и вы можете гарантировать, что в момент изменения ни один экземпляр не находится в коллекции `Set`.
- Каждый атрибут, имеющий в базе данных ограничение `UNIQUE`, является потенциальным кандидатом на роль бизнес-ключа. Помните, что точность бизнес-ключа должна быть достаточной для предотвращения коллизий.
- Любой атрибут, основанный на значении времени, такой как метка времени создания записи, обычно является хорошим компонентом бизнес-ключа, но погрешность метода `System.currentTimeMillis()` зависит от виртуальной машины и операционной системы. Мы рекомендуем буфер безопасности длиной в 50 миллисекунд, что может быть недостаточно точным, если атрибут, основанный на значении времени, является единственным атрибутом бизнес-ключа.
- Как часть бизнес-ключа можно использовать идентификатор в базе данных. Кажется, что это противоречит нашему предыдущему утверждению, но мы говорим не о значении идентификатора данной сущности, а об идентификаторе связанного экземпляра сущности. Например, потенциальным бизнес-ключом класса `Bid` (ставка) является идентификатор объекта `Item` (товар),

связанного со ставкой. Вы даже можете добавить ограничение уникальности, представляющее этот составной бизнес-ключ в схеме базы данных. Идентификатор связанного объекта `Item` никогда не изменится в течение жизненного цикла объекта `Bid`: конструктор класса `Bid` может требовать передачи объекта `Item`, уже находящегося в хранимом состоянии.

Если вы последуете нашему совету, у вас не должно возникнуть сложностей в выявлении хороших бизнес-ключей для всех классов, относящихся к бизнес-логике. Если вы столкнетесь со сложным случаем, попробуйте решить проблему без Hibernate. В конце концов, это лишь объектно-ориентированная проблема. Обратите внимание, что практически всегда будет ошибкой переопределять метод `equals()` в подклассах и добавлять в процедуру сравнения другие свойства. В подобном случае будет непросто удовлетворить требования к идентичности класса `Object` и равенству, которое должно быть в данном случае и симметричным, и транзитивным; и что важнее, на роль бизнес-ключа может не подойти ни один из кандидатов на естественные ключи в базе данных (свойства подкласса могут отображаться в другую таблицу). Чтобы узнать больше о реализации собственных методов сравнения, прочтите книгу Джошуа Блоха *Effective Java*, 2-е издание[1], обязательную для каждого программиста на Java.

Теперь класс `User` подготовлен для работы в отсоединенном состоянии: вы можете спокойно помещать в коллекцию `Set` экземпляры, загруженные в разных контекстах хранения. Далее мы рассмотрим несколько примеров, связанных с отсоединенным состоянием, и вы увидите несколько преимуществ этой идеи.

Иногда может понадобиться вручную отсоединить экземпляр сущности от контекста хранения.

10.3.3. Отсоединение экземпляров сущностей

Не обязательно ждать закрытия контекста хранения. Экземпляры сущности можно отсоединять вручную:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
User user = em.find(User.class, USER_ID);

em.detach(user);

assertFalse(em.contains(user));
```

Этот пример также демонстрирует операцию `EntityManager#contains()`, возвращающую `true`, если данный экземпляр находится в хранимом состоянии и управляется данным контекстом хранения.

Теперь можно работать со ссылкой `user` в отсоединенном состоянии. Многие приложения только читают данные и выводят их на экран после закрытия контекста хранения.

---

[1] *Блох Д.* Java. Эффективное программирование. 2-е изд. ISBN: 978-5-85582-348-6. Лори, 2016. – *Прим. ред.*

Модификация загруженного пользователя (`user`) после закрытия контекста хранения не окажет никакого влияния на его представление в базе данных. Тем не менее JPA позволяет перенести изменения в базу данных уже в новом контексте хранения.

10.3.4. Слияние экземпляров сущностей

Предположим, что вы извлекли экземпляр `User` в предыдущем контексте хранения, а теперь хотите его изменить и сохранить:

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
detachedUser.setUsername("johndoe");

tx.begin();
em = JPA.createEntityManager();
User mergedUser = em.merge(detachedUser);
mergedUser.setUsername("doejohn");

tx.commit();
em.close();
```

Ссылка detachedUser после слияния больше не нужна. mergedUser находится в хранимом состоянии

Выполнение операции UPDATE в базе данных

Рассмотрим графическое представление этого процесса (рис. 10.5). Это не так сложно, как кажется.

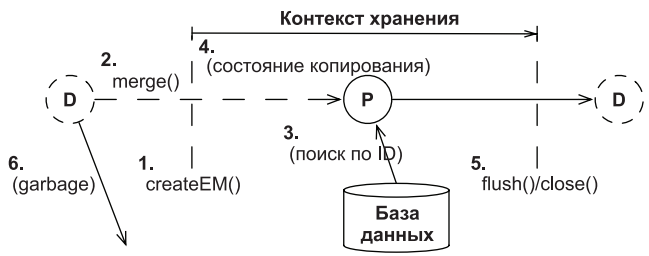


Рис. 10.5 ❖ Перемещение экземпляра в хранимое состояние в рамках единицы работы

Цель состоит в том, чтобы записать новое значение свойства `username` отсоединенного экземпляра `User`. Сначала, во время вызова `merge()`, Hibernate сравнит идентификаторы хранимого экземпляра из контекста хранения и отсоединенного экземпляра, с которым выполняется слияние.

В этом примере контекст хранения пуст; из базы данных ничего не было загружено. Поэтому Hibernate загрузит экземпляр с этим идентификатором из базы данных. Затем метод `merge()` скопирует отсоединенный экземпляр сущности *поверх* загруженного хранимого экземпляра. Другими словами, новое значение поля `username`, присвоенное отсоединенному объекту `User`, так же будет присвоено хранимому объекту `User`, который вернет метод `merge()` после слияния.

Теперь избавьтесь от ссылки на устаревшее и неактуальное отсоединенное состояние; ссылка `detachedUser` больше не представляет актуального состояния. Вы можете продолжить модификацию полученного объекта `mergedUser`; Hibernate выполнит единственную инструкцию `UPDATE`, когда будет выталкивать контекст хранения во время подтверждения транзакции.

Если в контексте хранения не будет найден ни один хранимый экземпляр с таким же идентификатором и поиск в базе данных по идентификатору не даст результатов, Hibernate создаст новый объект `User`. Затем выполнит копирование отсоединенного экземпляра поверх нового экземпляра и поместит его в базу данных при синхронизации контекста хранения с базой данных.

Если экземпляр, переданный в метод `merge()`, является не отсоединенным, а временным (у него отсутствует значение идентификатора), Hibernate создаст новый объект `User`, скопирует поверх временный объект `User`, сделает его хранимым и вернет обратно. Проще говоря, операция `merge()` может работать с отсоединенными *и* временными экземплярами сущностей. Hibernate всегда возвращает в результате хранимый экземпляр.

В архитектуре приложения, основанной на отсоединении и слиянии, можно не вызывать операцию `persist()`. Для сохранения данных можно выполнять слияние новых и отсоединенных экземпляров сущностей. Важное отличие заключается в возвращаемом текущем состоянии и способе переключения ссылок в коде приложения. Вы должны избавиться от ссылки `detachedUser` и после этого использовать только ссылку на текущее состояние `mergedUser`. Все остальные компоненты приложения, работающие со ссылкой `detachedUser`, должны переключиться на использование ссылки `mergedUser`.

Можно ли переподключить отсоединенный экземпляр?

В Hibernate-интерфейсе `Session` имеется метод для повторного присоединения `saveOrUpdate()`. Он принимает временный или отсоединенный экземпляр и ничего не возвращает. После выполнения операции переданный экземпляр будет находиться в хранимом состоянии, поэтому переключать ссылки не нужно. Hibernate выполнит инструкцию `INSERT`, если переданный экземпляр находился во временном состоянии, и `UPDATE`, если тот был отсоединен. Но вместо этого мы советуем полагаться на операцию слияния, поскольку она стандартизована и поэтому легче интегрируется с другими фреймворками. Кроме того, если отсоединенный экземпляр не был изменен, вместо `UPDATE` операция слияния выполнит только инструкцию `SELECT`. Если вам интересно, метод `saveOrUpdateCopy()` интерфейса `Session` делает то же самое, что и `merge()` в `EntityManager`.

Если понадобится удалить отсоединенный экземпляр, сначала нужно выполнить для него операцию слияния. Затем вызвать `remove()` для хранимого экземпляра, возвращаемого методом `merge()`.

Мы снова вернемся к обсуждению отсоединенного состояния и слияния в главе 18 и реализуем более сложное диалоговое взаимодействие между пользователем и системой, использующей эту стратегию.

10.4. Резюме

- Мы обсудили наиболее важные и некоторые дополнительные стратегии работы с экземплярами сущностей в JPA-приложениях.
- Вы познакомились с жизненным циклом экземпляров сущностей и узнали, как они становятся хранимыми, отсоединенными и удаленными.
- Самый важный интерфейс в JPA – это EntityManager.
- В большинстве приложений данные не сохраняются и не загружаются независимо. Как правило, Hibernate интегрируется в многопользовательское приложение с конкурентным доступом к базе данных в нескольких потоках.

Глава 11

Транзакции и многопоточность

В этой главе:

- основы определения системных транзакций и транзакций в базе данных;
- управление конкурентным доступом с помощью Hibernate и JPA;
- доступ к данным вне транзакции.

В этой главе мы, наконец, поговорим о транзакциях: как создавать единицы работы и управлять ими в многопоточном приложении. *Единица работы* – это атомарная группа операций. Транзакции позволяют определять границы единицы работы и помогают изолировать одну единицу работы от другой. В многопользовательском приложении также можно осуществлять параллельное выполнение нескольких единиц работы.

Прежде чем приступить к обсуждению поддержки многопоточности, мы сначала рассмотрим единицы работы на самом низком уровне – уровне системных транзакций и транзакций в базе данных. Познакомимся с API разграничения транзакций и способами определения единиц работы в коде на языке Java. Поговорим об изолированности и управлении параллельным доступом с помощью пессимистической и оптимистической стратегий.

В заключение мы рассмотрим некоторые специальные случаи и особенности JPA, связанные с доступом к базе данных без явного применения транзакций. Но для начала немного справочной информации.

Главные нововведения в JPA 2

- Добавлены новые режимы и исключения для работы с пессимистическими блокировками.
- Возможность выбора пессимистического или оптимистического режима блокировки с помощью интерфейса `Query`.
- Возможность выбора режима блокировки в вызове `EntityManager#find()`, `refresh()` или `lock()`. Также стандартизована подсказка о времени ожидания для режима пессимистической блокировки.

- При возбуждении одного из новых исключений – `QueryTimeoutException` или `LockTimeoutException` – можно не откатывать транзакцию.
- Контекст хранения теперь может находиться в рассинхронизированном состоянии с отключенным автоматическим сохранением контекста. Это позволяет добавлять изменения в очередь до присоединения к транзакции и отделить работу с `EntityManager` от транзакций.

11.1. Основы транзакций

Приложения часто требуют, чтобы несколько операций выполнялось как единое целое. Например, после завершения аукциона приложение CaveatEmptor должно выполнить три разные операции:

1) найти самую большую ставку для продаваемого товара;
2) взыскать с продавца стоимость проведения аукциона;
3) оповестить продавца и выигравшего покупателя.

Что случится, если не получится оплатить стоимость аукциона из-за ошибки во внешней системе для работы с кредитными картами? В бизнес-требованиях может быть указано, что либо все указанные действия должны завершиться успешно, либо ни одно из них. В таком случае эту совокупность задач можно назвать *транзакцией*, или единицей работы. Если хоть одна задача не выполнится, выполнение всей единицы работы должно окончиться неудачей.

11.1.1. Атрибуты ACID

Аббревиатура *ACID* расшифровывется как *atomicity* (атомарность), *consistency* (непротиворечивость), *isolation* (изолированность), *durability* (долговечность). *Атомарность* означает, что все операции в рамках транзакции выполняются как единое целое. Более того, транзакции позволяют нескольким пользователям параллельно работать с одними и теми же данными, не нарушая требования *непротиворечивости* данных (соответствующих правилам целостности в базе данных). Конкретная транзакция не должна быть видима для других транзакций, выполняющихся в это же время; они должны выполняться *изолированно* друг от друга. Изменения, сделанные в рамках транзакции, должны стать *постоянными*, даже если в системе произойдет сбой после успешного завершения транзакции.

Кроме того, транзакция должна быть *корректной*. Например, бизнес-правила определяют, что приложение должно взимать оплату с продавца только один раз, а не два. Это предположение разумно, но его невозможно выразить с помощью ограничений базы данных. Поэтому за корректность транзакции отвечает приложение, в то время как за непротиворечивость – база данных. Все вместе эти атрибуты определяют критерий *ACID*.

11.1.2. Транзакции в базе данных и системные транзакции

Мы уже упоминали о транзакциях в *системе* и *в базе данных*. Посмотрите еще раз на последний пример: во время выполнения единицы работы, завершающей аукцион,

мы можем определить самую большую ставку в базе данных. Затем, в рамках той же самой единицы работы, обратиться к внешней системе для списания платы с кредитной карты продавца. Эта транзакция охватывает несколько (под)систем с управляемыми подчиненными транзакциями, возможно охватывающими несколько ресурсов, таких как соединение с базой данных и внешняя платежная служба.

Транзакции в базах данных должны быть короткими, поскольку открытые транзакции потребляют ресурсы базы данных и потенциально могут ограничивать параллельный доступ из-за монопольных блокировок данных. Одна транзакция в базе данных, как правило, включает только один набор операций с базой данных.

Для выполнения всех операций в базе данных внутри системной транзакции необходимо определить границы конкретной единицы работы. Вы должны запустить транзакцию, а затем в какой-то момент подвердить изменения. Если произойдет ошибка (во время выполнения операции с базой данных или подтверждения транзакции), изменения следует откатить, чтобы данные остались в непротиворечивом состоянии. Этот процесс называется *определением границ транзакции* и в зависимости от используемого подхода может потребовать ручного управления. В целом границы, отмечающие начало и конец транзакции, могут задаваться как программно, так и декларативно.

11.1.3. Программные транзакции с JTA

В окружении Java SE границы транзакций определяются с помощью JDBC API. Транзакция начинается с вызова метода `setAutoCommit(false)` JDBC-объекта `Connection`, а завершается вызовом `commit()`. Все изменения можно откатить в любой момент вызовом метода `rollback()`.

В приложении, управляющем данными из нескольких систем, конкретная единица работы может обращаться к нескольким транзакционным ресурсам. В таком случае нельзя обеспечить атомарность только с помощью JDBC. Вам потребуется диспетчер транзакций, способный взаимодействовать с несколькими ресурсами в рамках одной системной транзакции. JTA стандартизирует управление системными и распределенными транзакциями, поэтому вам не нужно сильно беспокоиться о низкоуровневых деталях. Главным в JTA является интерфейс `UserTransaction` с методами `begin()` и `commit()`, запускающими и подтверждающими транзакции соответственно.

Прочие интерфейсы для определения границ транзакции

JTA поддерживает прекрасную абстракцию системы транзакций основных ресурсов, включающую также управление распределенными системными транзакциями. Большинство разработчиков полагает, что JTA можно применять только при работе с компонентами, работающими на сервере приложений Java EE. На сегодняшний день доступны независимые высококачественные реализации JTA, такие как Bitronix (используется в коде примеров) и Atomikos, которые легко устанавливаются в любом окружении Java. Эти решения можно считать пулами соединений с базой данных, работающими с JTA.

Вы должны использовать JTA, когда только возможно, и избегать нестандартных транзакционных API, таких как `org.hibernate.Transaction`, или очень ограниченного `javax.persistence.EntityTransaction`. Эти интерфейсы создавались во времена, когда JTA не мог полноценно работать вне среды выполнения контейнера EJB.

В разделе 10.2.1 мы обещали снова вернуться к транзакциям, уделив особое внимание обработке исключений. Ниже представлен полный код, включающий откат транзакций и обработку исключений.

Листинг 11.1 ❖ Типичная единица работы с границами транзакции

Файл: /examples/src/test/java/org/jpwh/test/simple/SimpleTransitions.java

```
EntityManager em = null;
UserTransaction tx = TM.getUserTransaction();
try {
    tx.begin();
    em = JPA.createEntityManager();  ← Управляется приложением

    // ...

    tx.commit();  ← Синхронизирует/выталкивает контекст хранения
} catch (Exception ex) {
    try {         ← Откат транзакции; обработка исключений
        if (tx.getStatus() == Status.STATUS_ACTIVE
            || tx.getStatus() == Status.STATUS_MARKED_ROLLBACK)
            tx.rollback();
    } catch (Exception rbEx) {
        System.err.println("Rollback of transaction failed, trace follows!");
        rbEx.printStackTrace(System.err);
    }
    throw new RuntimeException(ex);
} finally {
    if (em != null && em.isOpen())
        em.close();  ← Вы его создали, вы и закрывайте!
}
```

Самым сложным в этом примере является код обработки исключений; мы обсудим его чуть ниже, а сейчас разберемся, как взаимодействуют механизм управления транзакциями и `EntityManager`.

`EntityManager` использует стратегию отложенного выполнения; в предыдущей главе мы упоминали, что он не получает ни одного соединения с базой данных, пока не потребуется выполнить выражение SQL. То же самое верно для JTA: стоимость запуска и подтверждения транзакции окажется невысокой, если не обращаться ни к одному транзакционному ресурсу. Например, на сервере можно выполнить пустую единицу работы для каждого клиента, не потребляя никаких ресурсов и не удерживая блокировок в базе данных.

В момент создания объект `EntityManager` ищет внутри текущего потока выполнения незавершенную системную транзакцию JTA. Обнаружив ее, он *присоеди-*

няется к ней и начинает отслеживать события транзакции. Это означает, что вы всегда должны вызывать `UserTransaction#begin()` и `EntityManagerFactory#createEntityManager()` в одном потоке, если хотите, чтобы они соединились. Как мы уже объяснили в главе 10, по умолчанию Hibernate автоматически выталкивает контекст хранения во время подтверждения транзакции.

Если во время создания `EntityManager` не смог найти незавершенную транзакцию в том же потоке, он окажется в особом *рассинхронизированном* режиме. В этом режиме JPA не будет выталкивать контекст хранения автоматически. Позже мы еще поговорим об этом поведении; данная особенность JPA удобна для проектирования более сложных диалоговых взаимодействий.

Часто задаваемые вопросы: Правда ли, что транзакция с доступом только на чтение откатывается быстрее?

Если код в транзакции читает данные, но не изменяет их, можно ли просто откатить транзакцию вместо ее подтверждения? Будет ли такой код выполняться быстрее? Действительно, некоторые разработчики обнаружили, что при определенных условиях такой прием работает быстрее, и это убеждение распространилось среди сообщества. Мы проверяли это поведение на наиболее популярных СУБД и не нашли отличий. Также мы не нашли ни одного источника, подтверждающего различие в производительности. Нет причин использовать в СУБД неоптимальную реализацию операции завершения, которые требовали бы использовать не самый быстрый механизм завершения транзакции.

Всегда подтверждайте транзакции, а если подтверждение завершилось неудачей, откатывайте. Тем не менее в стандарте SQL есть выражение `SET TRANSACTION READ ONLY`. Мы советуем сначала узнать, поддерживает ли его ваша база данных и есть ли хоть какие-то преимущества с точки зрения производительности.

Диспетчер транзакций прервет транзакцию, если она выполняется слишком долго. Помните, что в нагруженной системе OLTP (оперативной обработки транзакций) длительность транзакции в базе данных должна быть минимальной. Время ожидания по умолчанию зависит от реализации JTA, например в Bitronix это 60 секунд. Это значение можно изменить перед началом транзакции вызовом `UserTransaction#setTransactionTimeout()`.

Мы должны еще обсудить обработку исключений, показанную в предыдущем примере.

11.1.4. Обработка исключений

Если обращение к экземпляру `EntityManager` или выталкивание контекста хранения во время подтверждения транзакции вызовет исключение, проверьте текущее состояние системной транзакции. В случае ошибки Hibernate отметит транзакцию как требующую отмены. Это означает, что единственным результатом этой транзакции будет отмена всех сделанных изменений. Поскольку запустили транзакцию вы, вы и должны проверить состояние `STATUS_MARKED_ROLLBACK`.

Транзакция также может находиться в состоянии `STATUS_ACTIVE`, если Hibernate не смог отметить ее как требующую отмены. В обоих случаях нужно вызвать `UserTransaction#rollback()`, чтобы предотвратить выполнение любых инструкций SQL, которые могли быть отправлены в базу данных в рамках данной единицы работы.

Любая операция JPA, включая выталкивание контекста хранения, способна возбудить исключение `RuntimeException`. Но такие методы, как `UserTransaction#begin()`, `commit()` и даже `rollback()`, возбуждают контролируемое исключение `Exception`. Исключение, возбуждаемое в случае отката транзакции, требует особого обращения: его нужно перехватить и записать в журнал; в противном случае исходное исключение, вызвавшее откат, окажется потерянным. После отката транзакции нужно повторно возбудить исходное исключение. Обычно в системе существует другой уровень перехватчиков, который в итоге обработает исключение, отобразив, например, экран с ошибкой или отправив сообщение команде технических специалистов. Гораздо труднее правильно обработать ошибку, возникшую во время отката транзакции; мы предлагаем записать ее в журнал и распространить дальше, поскольку ошибка при откате транзакции указывает на серьезную системную проблему.

Особенности Hibernate

Hibernate возбуждает типизированные исключения, наследующие `RuntimeException`, которые могут помочь в определении ошибки:

- наиболее общим типом является `HibernateException`. Нужно либо проверить сообщение об ошибке, либо выяснить причину, вызвав метод `getCause()` объекта-исключения;
- `JDBCException` представляет исключения, возбуждаемые внутренним слоем JDBC в Hibernate. Этот тип исключения всегда возникает из-за конкретного выражения SQL и позволяет получить выражение, вызвавшее ошибку, с помощью `getSQL()`. Информацию о внутреннем исключении, возбуждаемом соединением JDBC (драйвером JDBC), можно получить с помощью методов `getSQLException()` или `getCause()`, а код ошибки, характерный для базы данных и реализации, – с помощью `getErrorCode()`;
- Hibernate включает подтипы `JDBCException` и внутренний конвертер для преобразования кодов ошибок конкретной реализации, возвращаемых драйвером базы данных, в что-то более информативное. Встроенный конвертер может создавать экземпляры `JDBCConnectionException`, `SQLGrammarException`, `LockAcquisitionException`, `DataException` и `ConstraintViolationException` для наиболее важных диалектов баз данных, поддерживаемых Hibernate. Вы можете управлять диалектом своей базы данных или расширять его, а также подключать `SQLExceptionConverterFactory` для настройки этого преобразования.

Некоторые разработчики приходят в восторг от разнообразия хорошо детализированных типов исключений, поддерживаемых в Hibernate. Но это может

направить вас по ложному пути. К примеру, для целей валидации у вас может появиться желание перехватить `ConstraintViolationException`. Если вы забыли присвоить значение свойству `Item#name`, а его отображаемый столбец имеет ограничение `NOT NULL`, то при попытке вытолкнуть контекст хранения Hibernate возбудит это исключение. Почему бы не перехватить его, показать сообщение об ошибке (в зависимости от кода ошибки и текста сообщения) пользователям приложения и дать им возможность исправить ошибку? Такой подход имеет два больших недостатка.

Во-первых, отправка непроверенных данных в базу, чтобы узнать, какие окажутся неподходящими, является неверной стратегией для масштабируемых приложений. Приложение должно обеспечивать хотя бы минимальную валидацию данных. Во-вторых, исключения прерывают текущую единицу работы. Но это не соответствует тому, как пользователи будут интерпретировать ошибку валидации: они полагают, что все еще находятся внутри единицы работы. Обходить это несоответствие в коде трудно и неудобно. Мы советуем использовать детализированные типы исключений для отображения более информативных сообщений о (фатальных) ошибках, а не для валидации. Например, можно перехватить `ConstraintViolationException` и вернуть экран с сообщением: «Ошибка приложения: кто-то забыл проверить данные перед отправкой в базу. Пожалуйста, сообщите об этом программистам». Для других исключений можно показывать общий экран для отображения ошибок.

Подобное решение поможет вам во время разработки, а также любому инженеру службы поддержки, который должен быстро определить, является ли это ошибкой приложения (нарушение ограничений, выполнение неправильного кода SQL), или же СУБД находится под нагрузкой (невозможно получить блокировки). Для валидации у вас имеется универсальный фреймворк Bean Validation. Следуя единому набору правил, заданному с помощью аннотаций, Hibernate сможет проверить все ограничения предметной области и для конкретных записей на уровне пользовательского интерфейса, а также автоматически сгенерировать правила SQL DDL.

Теперь вы знаете, какие исключения перехватывать и где их ожидать. Возможно, вы задаете себе вопрос: что делать *после* перехвата исключения и отката системной транзакции? Исключения, возбуждаемые фреймворком Hibernate, фатальны. Это значит, что текущий контекст хранения нужно закрыть. Вам запрещается работать с экземпляром `EntityManager`, возбудившим исключение. Отобразите экран с ошибкой и/или запишите ошибку в журнал, а затем дайте пользователю возможность повторно начать диалоговое взаимодействие с системой, используя новую транзакцию и контекст хранения.

Но, как обычно, это еще не полная картина. Некоторые стандартизованные исключения не являются фатальными:

- `javax.persistence.NoResultException` – возбуждается, когда `Query` (запрос) или `TypedQuery` (типизированный запрос) выполняется вызовом метода `getSingleResult()`, а база данных не возвращает никаких результатов. Вызов

метода можно завернуть в код обработки исключений и продолжить работу с контекстом хранения. Текущая транзакция не будет отмечена как требующая отмены;

- `javax.persistence.NonUniqueResultException` – возбуждается, когда `Query` или `TypedQuery` выполняется вызовом `getSingleResult()`, а база данных возвращает несколько результатов. Вызов метода можно завернуть в код обработки исключений и продолжить работу с контекстом хранения. Hibernate не будет отмечать текущую транзакцию как требующую отмены;
- `javax.persistence.QueryTimeoutException` – возбуждается, когда запрос `Query` или `TypedQuery` выполняется слишком долго. Текущая транзакция не отмечается как требующая отмены. Можно попробовать снова выполнить запрос, если это уместно;
- `javax.persistence.LockTimeoutException` – возбуждается, когда пессимистическая блокировка не может быть получена. Это может произойти во время выталкивания контекста или явной установки блокировки (подробнее об этом рассказывается далее). Транзакция не отмечается как требующая отмены, и можно попытаться повторить операцию. Но помните, что частые обращения к базе данных, которая и так не справляется с нагрузкой, не улучшат ситуацию.

В этом списке отсутствует `javax.persistence.EntityNotFoundException`. Это исключение могут возбуждать методы `EntityManager#getReference()` и `refresh()`, а также `lock()`, с которыми вы познакомитесь далее в этой главе. Hibernate может возбудить его при попытке обратиться к ссылке (прокси-объекту) на экземпляр сущности, когда соответствующая запись в базе данных более недоступна. Это фатальное исключение: оно отмечает текущую транзакцию как требующую отмены – вы должны закрыть контекст хранения и избавиться от него.

Для программного определения границ транзакции требуется написать код, использующий специализированный интерфейс, такой как UserTransaction из `JTA`. Декларативное определение границ, напротив, не требует дополнительного кода.

11.1.5. Декларативное определение границ транзакции

В приложении Java EE можно *объявить*, когда работа должна выполняться внутри транзакции. Решение этой задачи берет на себя среда выполнения. Границы транзакции обычно задаются с помощью аннотаций управляемых компонентов (компонентов EJB, CDI и т. д.).

Для декларативного объявления границ транзакций компонентов EJB можно использовать старую аннотацию `@javax.ejb.TransactionAttribute`. Примеры вы найдете в разделе 18.2.1.

Для любого управляемого компонента Java EE можно применить более новую аннотацию общего назначения `@javax.transaction.Transactional`. Примеры вы найдете в разделе 19.3.1.

Все прочие примеры из этой главы работают в любом окружении Java SE и не требуют специальных контейнеров среды выполнения. Поэтому, начиная с данного момента, мы будем демонстрировать только код, определяющий границы транзакций программно, пока мы не начнем рассматривать примеры конкретных приложений Java EE.

Далее мы сосредоточимся на самом сложном из аспектов ACID – *изолированности* параллельно выполняющихся единиц работы.

11.2. Управление параллельным доступом

Базы данных (как и другие транзакционные системы) пытаются обеспечить *изолированность* транзакций: с точки зрения каждой конкретной транзакции никаких других транзакций как бы не существует. Традиционно СУБД обеспечивают изолированность с помощью блокировок. Транзакция может установить блокировку на конкретный элемент в базе данных, временно препятствуя другим транзакциям в получении доступа к этому элементу на чтение и/или на запись. Некоторые современные механизмы баз данных реализуют изолированность транзакций, используя управление параллельным доступом с помощью многоверсионности (Multiversion Concurrency Control, MVCC), который считается более масштабируемым решением. Мы будем обсуждать изолированность с точки зрения модели блокировок, но большая часть наших наблюдений также применима к MVCC.

Способ реализации управления многопоточностью в базе данных имеет огромное значение для приложения, использующего Java Persistence. Приложения пользуются гарантиями изолированности, предоставляемыми СУБД. Hibernate, к примеру, никогда ничего не блокирует в памяти. Преимущества такого подхода особенно очевидны, если учесть многие годы, в течение которых создатели баз данных работали над управлением многопоточностью. Кроме того, некоторые особенности Java Persistence могут обеспечить лучшие гарантии изолированности, чем в базе данных, независимо от того, применяете вы их явно или они действуют по умолчанию.

Мы будем знакомиться с управлением многопоточностью постепенно. Сначала исследуем нижний уровень: гарантии изолированности транзакций со стороны базы данных. Затем вы познакомитесь с пессимистическим и оптимистическим управлением транзакциями на уровне приложения в Java Persistence, а также с прочими гарантиями изолированности, которые можете дать Hibernate.

11.2.1. Многопоточность на уровне базы данных

Говоря об изолированности, нужно понимать, что изолированность либо есть, либо ее нет; в реальном мире не существует промежуточного состояния. В случае с транзакциями в базе данных за полную изолированность приходится платить высокую цену. Нельзя остановить мир, чтобы получить эксклюзивный доступ к данным в многопользовательской системе OLTP. Как следствие существует не-

сколько уровней изоляции, которые, естественно, ослабляют полную изоляцию, но повышают производительность и масштабируемость систем.

Особенности изоляции транзакций

Для начала рассмотрим некоторые явления, которые могут возникать при ослаблении полной изолированности транзакций. Стандарт ANSI SQL определяет несколько уровней изолированности, для каждого из которых данные явления допустимы.

Потерянное изменение (lost update) происходит, когда две транзакции обновляют один элемент данных, а затем последняя транзакция отменяется, вызывая потерю обоих изменений. Такое происходит в системах, где управление многопоточностью не реализовано, а параллельные транзакции не изолированы. Это показано на рис. 11.1.

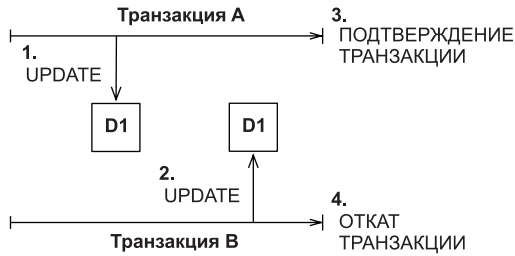


Рис. 11.1 ❖ Потерянное изменение: две транзакции обновляют одни и те же данные без изоляции

Чтение неподтвержденных изменений (dirty read) происходит, если транзакция читает изменения, сделанные другой транзакцией, которая еще не была подтверждена. Такая ситуация опасна, поскольку изменения, сделанные другой транзакцией, могут быть позже отменены, а первая транзакция может записать некорректные данные; см. рис. 11.2.

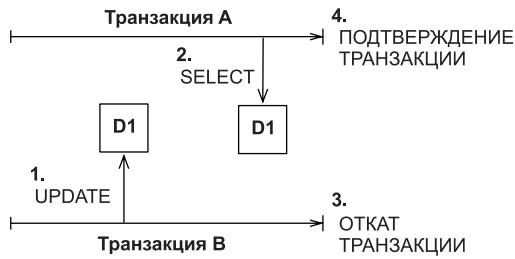


Рис. 11.2 ❖ Чтение неподтвержденных изменений: транзакция A читает неподтвержденные данные транзакции B

Неповторимое чтение (unrepeatable read) возникает, когда транзакция дважды читает элемент данных, и каждый раз данные оказываются в разном состоянии. Например, другая транзакция могла изменить элемент данных и оказаться подтвержденной между двумя чтениями, как показано на рис. 11.3.

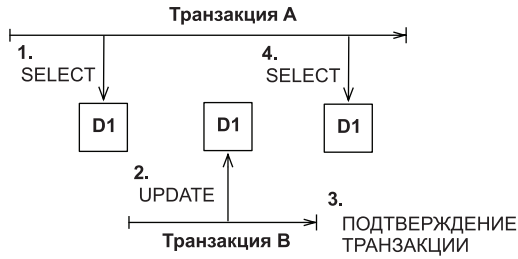


Рис. 11.3 ❖ Неповторимое чтение: транзакция A выполняет две операции чтения, возвращающие разные результаты

Особым случаем неповторимого чтения является проблема, когда побеждает *последнее изменение*. Представьте две параллельные транзакции, которые обе читают один элемент данных, как показано на рис. 11.4. Одна из них выполняет запись и подтверждается, а затем другая выполняет запись и тоже подтверждается. Изменения, сделанные первой транзакцией, будут потеряны. Эта проблема сильно расстраивает пользователей: изменения пользователя A будут перезаписаны без предупреждения, а пользователь B, возможно, принял решение, исходя из устаревшей информации.

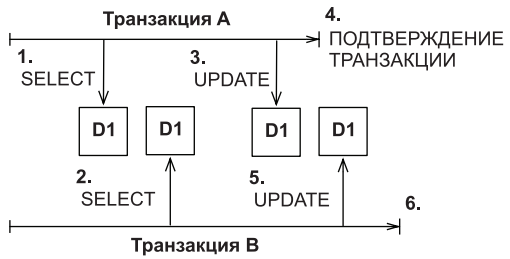


Рис. 11.4 ❖ Последняя транзакция побеждает: транзакция B затирает изменения, сделанные транзакцией A

Фантомное чтение (phantom read) возникает, когда транзакция выполняет запрос дважды, и второй результат включает либо больше данных, чем было видно первому запросу, либо меньше, поскольку что-то было удалено. Это не обязательно должен быть один запрос. Другая транзакция, которая вставляет или удаляет данные между выполнениями двух запросов, может вызвать ситуацию, показанную на рис. 11.5.

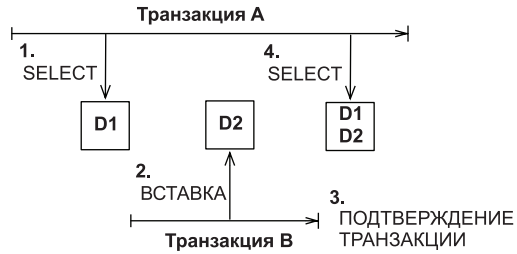


Рис. 11.5 ❖ Фантомное чтение: транзакция A читает измененные данные во время выполнения второй операции SELECT

Теперь, когда вы узнали обо всех неприятностях, которые могут случиться, мы можем определить уровни изолированности транзакций и узнать, какие из этих проблем они решают.

Уровни изолированности ANSI

Основные уровни изолированности транзакций определены в стандарте ANSI SQL, но они используются не только в базах данных SQL. JTA определяет точно такие же уровни изолированности, и вы будете использовать их для задания требуемого уровня изолированности транзакций. Увеличение уровня изолированности влечет увеличение стоимости и серьезное падение производительности и масштабируемости.

- *Чтение неподтвержденных данных* (read uncommitted) – система, работающая на данном уровне изолированности, допускает чтение неподтвержденных изменений, но не потерянные изменения. Транзакция не сможет сохранить изменения, если другая неподтвержденная транзакция уже изменила эту же запись. Однако любая транзакция может читать любые данные. СУБД может реализовать данный уровень изолированности с помощью монопольных блокировок на запись.
- *Чтение подтвержденных данных* (read committed) – система, работающая на данном уровне изолированности, допускает неповторимость чтения, но не позволяет читать неподтвержденные изменения. СУБД может работать на данном уровне, используя разделяемую блокировку на чтение и монопольную на запись. Транзакции, читающие данные, не блокируют доступа остальных транзакций, но неподтвержденная транзакция, выполняющая изменения, блокирует доступ к записи всех остальных транзакций.
- *Повторимое чтение* (repeatable read) – система, работающая на данном уровне изолированности, не допускает неповторимого чтения и чтения неподтвержденных изменений. Но фантомные чтения по-прежнему возможны. Транзакции, читающие данные, блокируют транзакции, выполняющие изменения, но не блокируют других читающих транзакций, а транзакции, выполняющие изменения, блокируют все остальные транзакции.

- *Упорядоченность* – самый строгий уровень изолированности. Имитирует последовательное выполнение, как если бы транзакции выполнялись одна за другой, а не параллельно. СУБД не может обеспечивать упорядоченность только за счет блокировок на уровне записей. Вместо этого СУБД должна поддерживать иной механизм, не позволяющий только что вставленной записи становиться видимой для транзакции, которая уже выполнила запрос, который вернул бы эту запись. Грубое решение заключается в монопольной блокировке целой таблицы базы данных после изменения для предотвращения фантомных чтений.

Способы реализации блокировок существенно различаются для разных СУБД; каждый разработчик проводит в жизнь свою стратегию. Чтобы узнать больше о механизме блокировок, их распространении (например, начиная уровнем записи и заканчивая страницами и целыми таблицами) и о том, какое воздействие оказывает каждый уровень изолированности на производительность и масштабируемость системы, нужно изучать документацию конкретной СУБД.

Конечно, неплохо знать, как определяются эти технические термины, но как это поможет в выборе уровня изолированности для приложения?

Выбираем уровень изолированности

Разработчики (в том числе и мы) часто не знают, какой уровень изолированности транзакций следует выбрать для приложения. В приложении с большим количеством потоков слишком высокий уровень изолированности навредит масштабируемости. Недостаточный уровень изолированности может вызывать появление скрытых, трудновоспроизводимых ошибок, которые не обнаруживаются, пока система не окажется под большой нагрузкой.

Обратите внимание, что далее мы будем использовать термин *оптимистическая блокировка* (с версионированием), значение которого объясняется далее в этой главе. Вы можете сейчас пропустить этот раздел и вернуться к нему позже, когда наступит время выбирать уровень изолированности вашего приложения. В конце концов, выбор уровня изолированности во многом зависит от конкретного сценария. Отнеситесь к последующему рассуждению как к рекомендациям, а не как к истине, высеченной в камне.

Hibernate старается быть как можно более прозрачным в том, что касается транзакционной семантики базы данных. Тем не менее кэширование в контексте хранения и версионирование влияют на эту семантику. Какой уровень изолированности целесообразно выбирать для приложений JPA?

Во-первых, никогда не используйте уровень изолированности, допускающий *чтение неподтвержденных данных*. Использовать неподтвержденные изменения одной транзакции в другой транзакции чрезвычайно опасно. Откат или ошибка одной транзакции повлияет на другие транзакции, выполняющиеся параллельно. Откат первой транзакции может откатить другие транзакции вместе с ней или даже повлиять на них так, что они оставят базу данных в некорректном состоянии (счет продавцу товара может быть выставлен дважды, что согласуется с пра-

вилами целостности в базе данных, но это некорректно). Кроме того, изменения, сделанные транзакцией, которая затем была отменена, могут быть подтверждены другой транзакцией, успевшей прочитать их и передать в базу данных!

Во-вторых, большинству приложений не требуется уровень изолированности, поддерживающий *упорядоченность*. Как правило, фантомное чтение обычно не вызывает проблем, и такой уровень изолированности плохо масштабируется. Немногие приложения используют уровень изолированности, поддерживающий упорядоченность; они скорее полагаются на выборочно используемые пессимистические блокировки, которые в определенных условиях фактически гарантируют последовательное выполнение операций.

Далее рассмотрим *повторимое чтение*. Этот уровень изолированности обеспечивает воспроизводимость результатов запроса в течение транзакции в базе данных. Это значит, что транзакция не прочитает подтвержденных изменений, даже если выполнит запрос несколько раз. Но фантомные чтения по-прежнему возможны: могут появляться новые строки, а те, которые существовали, могут исчезнуть, если другая транзакция параллельно подтвердит такие изменения. Иногда повторимое чтение желательно, но обычно требуется не в каждой транзакции.

В спецификации JPA по умолчанию используется уровень изолированности, обеспечивающий *чтение подтвержденных данных*. При его использовании придется иметь дело с неповторимым чтением, фантомным чтением и проблемой выигрыша последней подтвержденной транзакции.

Предположим, что для сущностей предметной модели вы применяете версионирование, поддерживаемое фреймворком Hibernate автоматически. Комбинация (обязательного) кэша контекста хранения и версионирования уже дает значительные преимущества уровня изолированности, обеспечивающего *повторимое чтение*. Кэш контекста хранения гарантирует изолированность состояния экземпляров сущностей, загруженных в одной транзакции, от изменений, сделанных в других транзакциях. Если в рамках единицы работы загрузить экземпляр сущности дважды, во второй раз искомая сущность будет взята из кэша контекста хранения, а не из базы данных. Следовательно, чтение *является* повторимым, и вы не увидите конфликтующих подтвержденных данных (вы все равно получите фантомные чтения, но иметь с ними дело, как правило, гораздо проще). Кроме того, при использовании версионирования побеждает *первая подтвержденная транзакция*. Следовательно, для большинства многопользовательских приложений JPA приемлемым является уровень изолированности, обеспечивающий *чтение подтвержденных изменений* для всех транзакций в базе данных при включенном версионировании.

Hibernate сохраняет уровень изолированности соединения с базой данных – он не меняет его. В большинстве приложений по умолчанию выбирается уровень изоляции, обеспечивающий *чтение подтвержденных данных*. Существует несколько способов изменения как уровня изолированности транзакций по умолчанию, так и настроек текущей транзакции.

Во-первых, вы можете проверить наличие глобального уровня изолированности транзакций СУБД в ее собственном файле настроек. Если СУБД поддерживает

стандартное SQL-выражение `SET SESSION CHARACTERISTICS`, можно выполнить его для установки настроек всех транзакций, запускаемых в данном конкретном *сеансе* работы с базой данных (имеется в виду конкретное подключение к базе данных, а не Hibernate-экземпляр `Session`). Также в SQL стандартизовано выражение `SET TRANSACTION`, устанавливающее уровень изолированности текущей транзакции. Наконец, JDBC `Connection` API включает метод `setTransactionIsolation()`, который (согласно документации) «пытается изменить уровень изолированности транзакции для данного подключения». В приложении Hibernate/JPA экземпляр JDBC `Connection` можно получить с помощью оригинального интерфейса `Session` (см. раздел 17.1).

Но если вы пользуетесь диспетчером транзакций JTA или даже простым пулом соединений JDBC, мы рекомендуем другой подход. Система управления транзакциями JTA, такая как Bitronix, используемая в примерах этой книги, позволяет устанавливать уровень изолированности транзакций по умолчанию отдельно для каждого соединения, полученного из пула. В Bitronix можно установить уровень изолированности по умолчанию вызовом `PoolingDataSource#setIsolationLevel()`. Более полную информацию ищите в документации к своей реализации источника данных, сервера приложений или пула соединений JDBC.

Начиная с этого момента, мы будем полагать, что ваши соединения с базой данных используют уровень изолированности по умолчанию, обеспечивающий *чтение подтвержденных данных*. Время от времени конкретная единица работы в приложении может потребовать другого, обычно более строгого уровня изолированности. Вместо изменения уровня изолированности целой транзакции используйте Java Persistence API, чтобы получить дополнительные блокировки для соответствующих данных. Обычно хорошо детализированные блокировки лучше масштабируются в приложениях с большим количеством потоков. JPA поддерживает оптимистическую проверку версий и пессимистическую блокировку на уровне базы данных.

11.2.2. Оптимистическое управление параллельным доступом

Оптимистическое управление параллельным доступом подходит в случаях, когда изменения вносятся редко и в рамках единицы работы допустимо позднее обнаружение конфликтов. Также JPA поддерживает автоматическую проверку версий в процедуре оптимистического обнаружения конфликтов.

Во-первых, вы должны включить версионирование, поскольку по умолчанию оно отключено: именно поэтому всегда будет побеждать *последняя зафиксированная транзакция*. Большинству многопользовательских приложений, и особенно веб-приложений, следует полагаться на версионирование всех экземпляров с аннотацией `@Entity`, которые могут изменяться в многопоточной среде, и использовать более дружественную для пользователя стратегию, при которой побеждает *первая подтвержденная транзакция*.

Предыдущий раздел был достаточно скучным, зато теперь мы займемся практикой. После включения автоматической проверки версий вы узнаете, как работает ручная проверка и когда ее нужно использовать.

Включаем версионирование

Версионирование включается с помощью аннотации `@Version` перед специальным дополнительным свойством класса сущности, как показано далее.

Листинг 11.2 ❖ Включаем версионирование отображаемой сущности

Файл: /model/src/main/java/org/jpwh/model/concurrency/version/Item.java

```
@Entity
public class Item implements Serializable {

    @Version
    protected long version;

    // ...
}
```

В этом примере каждому экземпляру сущности приваивается номер версии. Он отображается на дополнительный столбец в таблице `ITEM`. Как обычно, имя столбца по умолчанию совпадает с именем свойства, в данном случае `VERSION`. Настоящие имена свойства и столбца не важны – если `VERSION` является зарезервированным словом в СУБД, используйте другие имена.

Вы можете добавить в класс метод `getVersion()`, но у вас не должно быть метода записи, и приложение не должно менять значения поля. Hibernate меняет номер версии автоматически: он будет увеличивать его каждый раз, когда объект `Item` будет оказываться в измененном состоянии во время выталкивания контекста хранения. Версия – это простой счетчик, не обладающий никакой полезной семантикой, кроме той, что относится к управлению параллельным доступом. Можно использовать типы `int`, `Integer`, `short`, `Short` или `Long` вместо `long`; если значение версии достигнет предельного для данного типа значения, Hibernate начнет отсчет с нуля.

После увеличения номера версии экземпляра `Item`, чье состояние изменилось, Hibernate сравнит версии во время выполнения SQL-выражений `UPDATE` и `DELETE`. Например, представьте, что в рамках единицы работы вы загрузили экземпляр `Item` и поменяли его имя, как показано в листинге 11.3.

Листинг 11.3 ❖ Hibernate автоматически увеличивает номер версии и выполняет сравнение

Файл: /examples/src/test/java/org\jpwh/test/concurrency/Versioning.java

```
tx.begin();
em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);  <-- ❶ Поиск по идентификатору
// select * from ITEM where ID = ?

assertEquals(item.getVersion(), 0);  <-- ❷ Версия экземпляра: 0

item.setName("New Name");

em.flush();  <-- ❸ Выталкивает контекст хранения
// update ITEM set NAME = ?, VERSION = 1 where ID = ? and VERSION = 0
```

❶ Поиск экземпляра сущности по идентификатору загружает текущую версию из базы данных с помощью оператора `SELECT`.
❷ Текущая версия экземпляра `Item` равна 0.
❸ Выталкивая контекст хранения, Hibernate обнаружит изменения экземпляра `Item` и увеличит версию до 1. SQL-выражение `UPDATE` теперь выполняет проверку, сохраняя новый номер версии, только если в базе данных он еще равен 0.

Обратите внимание на выражения SQL, в частности `UPDATE` и его предложение `WHERE`. Это изменение будет успешным лишь в том случае, если в базе данных *существует* запись, для которой истинно выражение `VERSION = 0`. JDBC вернет фреймворку Hibernate количество измененных записей; если в результате получится ноль, это означает, что такой записи в таблице `ITEM` больше нет или ее версия отличается от `0`. Hibernate обнаружит конфликт во время выталкивания контекста и возбудит исключение `javax.persistence.OptimisticLockException`.

Представьте теперь двух пользователей, выполняющих эту единицу работы параллельно, как показано на рис. 11.4. Подтверждение транзакции первым пользователем обновит имя экземпляра `Item` и запишет в базу данных увеличенный номер версии, равный `1`. Подтверждение транзакции (и выталкивание контекста) вторым пользователем завершится неудачей, поскольку выполняемое им выражение `UPDATE` не найдет в базе данных записи с версией `0`. Номер версии в базе данных равен `1`. Таким образом, победит *первая зафиксированная транзакция*, а вы сможете перехватить исключение `OptimisticLockException` и обработать его. Например, можно показать второму пользователю следующее сообщение: «Данные, с которыми вы работали, были изменены другим пользователем. Пожалуйста, начните выполнение задачи повторно с измененными данными. Нажмите кнопку “Начать сначала” для продолжения».

Какие изменения приводят к увеличению номера версии сущности? Hibernate увеличивает номер версии каждый раз, обнаруживая изменения в состоянии экземпляра сущности. К ним относятся все изменения свойств сущности с типами-значениями независимо от того, являются они скалярными (как поля типа `String` или `int`), встроенными (как `Address`) или коллекциями. Исключение составляют коллекции ассоциаций `@OneToMany` и `@ManyToMany` с доступом только на чтение, настроенным при помощи `mappedBy`. Добавление или удаление элементов из этих коллекций не увеличивает номера версии экземпляра сущности-владельца. Вы должны понимать, что ничего из этого в JPA не стандартизовано – работая с общей базой данных, не нужно полагаться, что две разные реализации JPA следуют одним и тем же правилам.

Версионирование в общей базе данных

Если к вашей базе данных обращается сразу несколько приложений и не все они используют алгоритм версионирования Hibernate, вы можете столкнуться с проблемами параллельного доступа. Эта проблема легко решается с использованием триггеров и хранимых процедур на уровне базы данных: триггер `INSTEAD OF` может вызывать хранимую

процедуру в каждой операции UPDATE; она будет выполняться вместо изменения. Внутри процедуры можно проверить, увеличило ли приложение номер версии записи; если версия не обновлялась или столбец версии не был включен в операцию изменения, значит, это выражение было сформировано не в приложении Hibernate. В таком случае процедура могла бы увеличить номер версии перед выполнением UPDATE.

Если вы не хотите увеличивать номер версии экземпляра сущности при изменении конкретных полей, их нужно отметить аннотацией `@org.hibernate.annotations.OptimisticLock(excluded = true)`. Вам также может не понравиться наличие дополнительного столбца VERSION в схеме базы данных. С другой стороны, у вас уже могут иметься поле с отметкой времени последнего изменения в классе сущности и соответствующий столбец в базе данных. Hibernate может проверять версии, используя отметки времени вместо дополнительного поля счетчика.

Версионирование с помощью отметок времени

Если в схеме вашей базы данных уже имеется столбец с отметкой времени, такой как LASTUPDATED или MODIFIED_ON, вместо обычного счетчика для автоматической проверки версии можно воспользоваться им.

Листинг 11.4 ❖ Версионирование с помощью отметок времени

Файл: /model/src/main/java/org/jpwh/model/concurrency/versiontimestamp/Item.java

```
@Entity
public class Item {

    @Version
    // Необязательно: @org.hibernate.annotations.Type(type = "dbtimestamp")
    protected Date lastUpdated;

    // ...
}
```

В этом примере столбец LASTUPDATED отображается в свойство типа `java.util.Date`; также Hibernate может работать с типом `Calendar`. Но использование свойств этих типов для поддержки версионирования не регламентируется стандартом JPA; согласно JPA переносимым решением является только тип `java.sql.Timestamp`. Это не самый лучший вариант, поскольку придется импортировать этот JDBC-класс в предметную модель. Вы должны стараться изолировать такие детали реализации от классов предметной модели, чтобы последние можно было протестировать, скомпилировать кросс-компилятором (например, в JavaScript с помощью GWT), сериализовать и десериализовать в как можно большем количестве окружений.

Теоретически версионирование с помощью отметки времени является чуть менее безопасным, поскольку две параллельно выполняющиеся транзакции могут обновить и сохранить тот же экземпляр `Item` в одну миллисекунду; это усугубляется и тем фактом, что обычно JVM не поддерживает точности до миллисекунды (чтобы узнать гарантированную точность, сверьтесь с документацией к вашей

JVM и операционной системе). Кроме того, получение текущего времени в JVM может быть небезопасным в кластерном окружении, где системное время узлов может быть не синхронизировано, или синхронизация времени может быть не настолько точной, насколько этого требуют транзакции.

Особенности Hibernate

Вы можете переключиться на использование текущего времени сервера с баз данных, добавив аннотацию `@org.hibernate.annotations.Type(type="dbtimestamp")` перед свойством версии. В этом случае Hibernate будет запрашивать текущее время у базы данных перед обновлением, например с помощью выражения `call current_timestamp()` для базы данных H2. Это позволит использовать единый ресурс для синхронизации времени. Не все SQL-диалекты Hibernate поддерживают это, поэтому загляните в исходный код своего диалекта, чтобы узнать, переопределяет ли он метод `getCurrentTimestampSelectString()`. Помимо этого, к базе данных происходит чрезмерное количество обращений при каждом увеличении номера версии.

В новых проектах мы советуем использовать версионирование со счетчиком. Если вы работаете с унаследованной схемой базы данных или уже существующими классами Java, когда невозможно добавить номер версии или отметку времени с соответствующими столбцами, то у Hibernate есть для вас альтернативное решение.

Особенности Hibernate

Версионирование без номера версии или метки времени

Если у вас нет столбцов для хранения версии или отметки времени, Hibernate все же может обеспечить автоматическое версионирование. Альтернативная реализация версионирования сравнивает текущее состояние в базе данных с неизменными значениями хранимых свойств на момент загрузки экземпляра сущности (или на момент последнего выталкивания контекста хранения).

Эту функциональность можно подключить с помощью оригинальной Hibernate-аннотации `@org.hibernate.annotations.OptimisticLocking`:

Файл: model\src\main\java\org\jpwh\model\concurrency\versionall\Item.java

```
@Entity
@org.hibernate.annotations.OptimisticLocking(
    type = org.hibernate.annotations.OptimisticLockType.ALL)
@org.hibernate.annotations.DynamicUpdate
public class Item {

    // ...
}
```

Также нужно включить динамическое формирование SQL-выражений UPDATE, добавив аннотацию @org.hibernate.annotations.DynamicUpdate, как объяснялось в разделе 4.3.2.

Теперь для сохранения изменений экземпляра Item Hibernate выполнит следующее выражение SQL:

```
update ITEM set NAME = 'New Name'
    where ID = 123
        and NAME = 'Old Name'
        and PRICE = '9.99'
        and DESCRIPTION = 'Some item for auction'
        and ...
        and SELLER_ID = 45
```

В предложении WHERE Hibernate перечислит все столбцы с их последними известными значениями. Если какая-то параллельная транзакция изменила любое из этих значений или даже удалила запись, это выражение ничего не обновит и вернет ноль. Затем, во время выталкивания контекста, Hibernate возбудит исключение.

С другой стороны, Hibernate может использовать для ограничения только измененные поля (в данном примере это только NAME), если выбрать OptimisticLockType.DIRTY. Это значит, что две единицы работы могут параллельно изменить один экземпляр Item, и Hibernate обнаружит конфликт, только если обе изменят одно и то же поле с типом-значением (или значение внешнего ключа). Предложение WHERE из последнего выражения SQL сократится до where ID = 123 and NAME = 'Old Name'. Кто-то другой может параллельно изменить цену, но Hibernate не заметит этого. Исключение javax.persistence.OptimisticLockException возникнет, только если приложение параллельно изменит значение имени.

Как правило, проверка только изменившихся полей – не лучшая стратегия для бизнес-сущностей. Возможно, не совсем правильно менять цену товара, когда поменялось лишь описание!

Также эта стратегия не подходит для работы с отсоединенным состоянием и слиянием: при слиянии отсоединенного экземпляра с новым контекстом хранения «старые» значения неизвестны. Для оптимистического управления параллельным доступом отсоединенному экземпляру сущности понадобится номер версии или отметка времени.

Автоматическое версионирование в Java Persistence позволяет избежать потерянных изменений, когда две параллельные транзакции пытаются подтвердить изменения одного и того же элемента информации. Также версионирование вручную позволит получить дополнительные гарантии изоляции, если они вам понадобятся.

Проверка версии вручную

Рассмотрим сценарий, требующий повторного чтения информации из базы данных; представьте, что в систему аукционов добавлены категории и каждый то-

вар (Item) относится к некоторой категории (Category). Это обычное отображение @ManyToOne связи Item#category.

Предположим, вам понадобилось найти сумму цен на все товары в нескольких категориях. Для суммирования потребуется запросить все товары в каждой категории. А теперь представьте, что произойдет, если кто-то переместит некоторый товар из одной категории в другую, пока вы будете выполнять запросы, последовательно обращаясь ко всем товарам и категориям? Используя уровень изолированности, при котором возможно чтение только подтвержденных данных, один и тот же экземпляр Item может быть обработан дважды!

Чтобы добиться повторимого чтения для запроса получения всех товаров в каждой категории, в JPA-интерфейсе Query объявлен метод setLockMode(). Взгляните на процедуру в листинге 11.5.

Листинг 11.5 ❖ Проверка версии во время выталкивания контекста для обеспечения повторимого чтения

Файл: /examples/src/test/java/org/jpwh/test/concurrency/Versioning.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

BigDecimal totalPrice = new BigDecimal(0);
for (Long categoryId : CATEGORIES) {

    List<Item> items =  ❶ Запрос, использующий оптимистический (OPTIMISTIC) режим блокировки
        em.createQuery("select i from Item i where i.category.id = :catId")
            .setLockMode(LockModeType.OPTIMISTIC)
            .setParameter("catId", categoryId)
            .getResultList();

    for (Item item : items)
        totalPrice = totalPrice.add(item.getBuyNowPrice());
}

tx.commit();  ❷ Выполняет инструкцию SELECT во время выталкивания контекста
em.close();

assertEquals(totalPrice.toString(), "108.00");
```

❶ Для каждой категории (Category) выбираются все товары (Item) в режиме блокировки OPTIMISTIC. Теперь Hibernate знает, что во время выталкивания контекста нужно проверить каждый экземпляр Item.

❷ Для каждого объекта Item, который был ранее загружен в блокирующем запросе, во время выталкивания контекста Hibernate выполнит инструкцию SELECT. Он проверит версии во всех записях в таблице ITEM. Если хотя бы в одной записи версия будет отличаться или она была удалена, фреймворк возбудит исключение OptimisticLockException.

И пусть вас не вводит в заблуждение терминология, связанная с *блокировками*: спецификация JPA не объясняет, как должен быть реализован каждый режим блокировок (LockModeType); при выборе режима OPTIMISTIC Hibernate выполняет проверку версий. Блокировки при этом не используются. Вы должны включить

версионирование для класса сущности `Item`, как объяснялось ранее, в противном случае вы не сможете использовать оптимистический режим блокировок (`LockModeType`) в Hibernate.

При проверке версии вручную Hibernate никак не оптимизирует выражения `SELECT`: если вам потребуется сложить цены 100 товаров, во время выталкивания контекста вы получите 100 дополнительных запросов. Как мы покажем далее в этой главе, для этой конкретной ситуации лучше подойдет пессимистический подход.

Часто задаваемые вопросы:

Почему кэш контекста хранения не может решить эту проблему?

Запрос, выбирающий все товары из конкретной категории, возвращает данные в виде объекта `ResultSet` (набора результатов). Затем Hibernate просматривает значения первичных ключей в этом наборе и сначала пытается получить остальные данные для каждого экземпляра `Item` в кэше контекста: он проверяет, был ли загружен экземпляр `Item` с таким идентификатором. Однако для процедуры из примера этот кэш будет бесполезен: если параллельная транзакция переместит товар из одной категории в другую, такой товар может войти в несколько разных коллекций `ResultSet`. Hibernate выполнит поиск по идентификатору в своем кэше и скажет: «Да я ведь уже загрузил этот экземпляр `Item`; буду использовать то, что уже загружено в память». Hibernate даже не узнает, что категория товара изменилась или товар появился второй раз в другом наборе результатов. В данном случае повторное чтение в контексте хранения скрывает параллельное изменение данных. Чтобы узнать, не изменились ли данные, нужно проверять версию вручную.

Как показано в предыдущем примере, интерфейс `Query` принимает параметр `LockModeType`. Явное указание режима блокировки также поддерживается интерфейсами `TypedQuery` и `NamedQuery` с помощью такого же метода `setLockMode()`.

В JPA доступен дополнительный оптимистический режим блокировки, который всегда увеличивает версию сущности.

Принудительное увеличение номера версии

Что произойдет, если два пользователя разместят ставку на один и тот же товар одновременно? Когда пользователь размещает ставку, приложение должно выполнить несколько действий.

1. Найти в базе данных самую большую ставку (`Bid`) за товар (`Item`).
2. Сравнить текущую ставку с самой большой ставкой (`Bid`); если новая ставка (`Bid`) окажется выше, сохранить его в базу данных.

Между этими двумя шагами может возникнуть ситуация гонки. Если между чтением самой большой ставки (`Bid`) и записью новой ставки (`Bid`) будет сделана еще одна ставка (`Bid`), вы об этом не узнаете. Это невидимый конфликт; не поможет даже добавление версионирования для класса `Item`. Во время этой операции экземпляр `Item` не обновляется. Только принудительное увеличение номера версии экземпляра `Item` позволит обнаружить конфликт.

Листинг 11.6 ❖ Принудительное увеличение номера версии экземпляра сущности

Файл: /examples/src/test/java/org/jpwh/test/concurrency/Versioning.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(  ← ❶ Вынуждает Hibernate увеличить номер версии
    Item.class,
    ITEM_ID,
    LockModeType.OPTIMISTIC_FORCE_INCREMENT
);

Bid highestBid = queryHighestBid(em, item);
try {
    Bid newBid = new Bid(  ← ❷ Сохраняет экземпляр Bid
        new BigDecimal("44.44"),
        item,
        highestBid
    );
    em.persist(newBid);
} catch (InvalidBidException ex) {  ← ❸ Проверка ставки
}

tx.commit();  ← ❹ Выполнение INSERT для экземпляра Bid
em.close();
```

❶ Метод `find()` принимает параметр типа `LockModeType`. Режим `OPTIMISTIC_FORCE_INCREMENT` требует от Hibernate увеличить версию полученного экземпляра `Item` после загрузки, даже если он не изменялся в рамках единицы работы.

❷ Этот код сохраняет новый экземпляр `Bid`, никак не влияя на состояние экземпляра `Item`. Происходит вставка новой записи в таблицу `BID`. Без принудительного увеличения номера версии в экземпляре `Item` Hibernate не сможет обнаружить изменения наибольшей ставки, сделанного параллельно.

❸ Для проверки новой ставки используется контролируемое исключение. Новая ставка должна превышать самую большую на текущий момент.

❹ Во время выталкивания контекста хранения Hibernate выполнит инструкцию `INSERT` для нового экземпляра `Bid`, а также инструкцию `UPDATE` с проверкой версии экземпляра `Item`. Если кто-то параллельно изменил экземпляр `Item` или сделал новую ставку с помощью этой процедуры, Hibernate возбудит исключение.

Параллельное размещение ставок, очевидно, часто будет происходить в системе для аукционов. Увеличение номера версии вручную может пригодиться во многих ситуациях, когда приходится вставлять или изменять данные и требуется увеличивать версию некоторого аггрегирующего экземпляра.

Обратите внимание, что если вместо связи `@ManyToOne` сущностей `Bid#item` установить связь `Item#bids` с помощью аннотации `@ElementCollection`, добавление ставки (`Bid`) в коллекцию *увеличит* номер версии экземпляра `Item`. В таком случае принудительное увеличение версии не потребуется. Возможно, вам захочется заново просмотреть обсуждение неоднозначного отношения родитель/потомок и как в ORM работают агрегаты и отношения композиции в разделе 7.3.

До сих пор мы говорили только об оптимистическом управлении параллельным доступом: мы полагали, что параллельные изменения будут редкими, и заранее не предотвращали параллельного доступа, обнаруживая конфликты позднее. Иногда известно, что конфликты могут случаться часто и на некоторые данные нужно устанавливать монопольную блокировку. В таком случае применяется пессимистический подход.

11.2.3. Явные пессимистические блокировки

Давайте переделаем процедуру из предыдущего раздела с помощью пессимистической блокировки, заменив ею оптимистическую проверку версии. Итак, нам снова требуется найти сумму всех цен на товары в нескольких категориях. Программный код останется прежним, как было в листинге 11.5, изменится только режим блокировки (`LockModeType`).

Листинг 11.7 ❖ Пессимистическая блокировка данных

Файл: /examples/src/test/java/org/jpwh/test/concurrency/Locking.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

BigDecimal totalPrice = new BigDecimal(0);
for (Long categoryId : CATEGORIES) {

    List<Item> items =  ← ❶ Запрашивает все экземпляры Item
        em.createQuery("select i from Item i where i.category.id = :catId")
            .setLockMode(LockModeType.PESSIMISTIC_READ)
            .setHint("javax.persistence.lock.timeout", 5000)
            .setParameter("catId", categoryId)
            .getResultList();
    for (Item item : items)  ← ❷ Успешное выполнение означает получение монопольной блокировки
        totalPrice = totalPrice.add(item.getBuyNowPrice());
}

tx.commit();  ← ❸ Блокировка снимается
em.close();

assertEquals(totalPrice.compareTo(new BigDecimal("108")), 0);
```

❶ Для каждой категории (`Category`) нужно выбрать все товары (`Item`) в режиме `PESSIMISTIC_READ`. Hibernate заблокирует записи в базе данных с помощью запроса SQL. Если прежде другая транзакция наложила конфликтующую блокировку, подождем 5 секунд. Если установить блокировку невозможно, запрос возбудит исключение.

❷ Если запрос выполнился успешно, можно быть уверенным, что монопольная блокировка получена и никакая другая транзакция не сможет ни получить монопольную блокировку, ни модифицировать данных, пока текущая транзакция не будет подтверждена.

❸ После подтверждения транзакции все блокировки снимаются.

Спецификация JPA указывает, что режим блокировки `PESSIMISTIC_READ` гарантирует повторимость чтения. Также JPA указывает, что режим `PESSIMISTIC_WRITE` предоставляет дополнительные гарантии: в дополнение к повторимому чтению

реализация JPA должна предоставить последовательный доступ к данным, чтобы сделать невозможными фантомные чтения.

Только реализация JPA решает, следовать ли этим требованиям. В обоих режимах Hibernate добавляет в запрос SQL предложение «for update». Таким способом устанавливается блокировка записи на уровне базы данных. Тип блокировки, используемой Hibernate, зависит от параметра `LockModeType` и диалекта базы данных.

Например, для H2 запрос примет следующий вид: `SELECT * FROM ITEM ... FOR UPDATE`. Поскольку H2 поддерживает лишь один тип монопольных блокировок, для всех пессимистических режимов Hibernate сгенерирует одинаковый код SQL.

PostgreSQL, с другой стороны, поддерживает разделяемые блокировки на чтение: режим `PESSIMISTIC_READ` добавит в запрос SQL предложение `FOR SHARE`. `PESSIMISTIC_WRITE` использует монопольную блокировку на запись, добавляя предложение `FOR UPDATE`.

В MySQL параметр `PESSIMISTIC_READ` превращается в `LOCK IN SHARE MODE`, а `PESSIMISTIC_WRITE` – в `FOR UPDATE`. Проверьте диалект вашей базы данных. Настройка осуществляется при помощи методов `getReadLockString()` и `getWriteLockString()`.

Пессимистическая блокировка в JPA длится столько же, сколько транзакция в базе данных. Поэтому монопольную блокировку нельзя использовать для предотвращения параллельного доступа более чем на время одной транзакции. Если блокировка в базе данных не может быть получена, возбуждается исключение. Сравните это поведение с оптимистическим подходом, когда Hibernate возбуждает исключение во время подтверждения транзакции, а не во время выполнения запроса. Используя пессимистическую стратегию, можно безопасно читать и записывать данные, если блокирующий запрос выполнился успешно. Используя оптимистический подход, приходится надеяться на лучшее, но можно неприятно удивиться позже, во время подтверждения транзакции.

Автономные блокировки

Пессимистические блокировки в базе данных действуют только в течение одной транзакции. Также возможны другие реализации блокировок, например блокировка в памяти или так называемая *таблица блокировок* в базе данных. Такие типы блокировок называются автономными.

Пессимистическая блокировка, длящаяся дольше одной транзакции, обычно негативно влияет на производительность; каждое обращение к данным требует дополнительных проверок с помощью глобального синхронизированного диспетчера блокировок. С другой стороны, оптимистическая блокировка является прекрасной стратегией управления параллельным доступом при длительных диалоговых взаимодействиях (как вы увидите в следующей главе) и обладает прекрасной производительностью. В зависимости от стратегии разрешения конфликтов – т. е. действий после обнаружения конфликта – пользователи приложения будут довольны настолько, насколько это позволит блокирование параллельного доступа. Также им бы понравилось, что приложение не блокирует доступа к определенным экранам с данными, когда другие просматривают те же самые данные.

Есть возможность задать время, в течение которого база данных будет ожидать получения блокировки и приостанавливать выполнение запроса, применяя рекомендацию `javax.persistence.lock.timeout`. Как обычно бывает с рекомендациями, Hibernate может ее проигнорировать в зависимости от реализации базы данных. Например, H2 не поддерживает времени ожидания блокировки для отдельного запроса – только общее время ожидания для каждого соединения (по умолчанию 1 сек). В некоторых диалектах, таких как PostgreSQL и Oracle, время ожидания блокировки, равное `0`, добавляет в запрос SQL предложение `NOWAIT`.

Мы показали, как использовать рекомендацию со временем ожидания блокировки для интерфейса `Query`. Вы также можете использовать рекомендацию со временем ожидания в операциях `find()`:

Файл: /examples/src/test/java/org/jpwh/test/concurrency/Locking.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Map<String, Object> hints = new HashMap<String, Object>();
hints.put("javax.persistence.lock.timeout", 5000);

Category category =
    em.find(
        Category.class,
        CATEGORY_ID,
        LockModeType.PESSIMISTIC_WRITE,
        hints
    );

category.setName("New Name");

tx.commit();
em.close();
```

Выполнит запрос SELECT ... FOR UPDATE WAIT 5000, если диалект поддерживает подсказки

Когда блокировка не может быть получена, Hibernate возбудит исключение `javax.persistence.LockTimeoutException`, или `javax.persistence.PessimisticLockException`. Получив исключение `PessimisticLockException`, приложение должно откатить транзакцию и прекратить выполнение единицы работы. Исключение `LockTimeoutException`, наоборот, не является фатальным, как объяснялось в разделе 11.1.4. Тип исключения зависит от диалекта SQL. Например, поскольку H2 не поддерживает времени ожидания блокировки для отдельного выражения, вы всегда будете получать `PessimisticLockException`.

Вы можете использовать оба режима – `PESSIMISTIC_READ` и `PESSIMISTIC_WRITE`, даже если не используете версионирования сущностей. Они будут преобразованы в выражения SQL с блокировками на уровне базы данных.

Однако специальный режим `PESSIMISTIC_FORCE_INCREMENT` требует версионирования. В Hibernate этот режим использует блокировку `FOR UPDATE NOWAIT` (или ту, что поддерживает диалект базы данных; посмотрите реализацию метода `getForUpdateNowaitString()`). Затем, сразу же после выполнения запроса, Hibernate увеличивает номер версии и выполняет операцию `UPDATE` для (!) каждого получен-

ного экземпляра сущности. Это сообщит всем параллельным транзакциям, что вы обновили эти строки, даже если никакие данные не изменялись. Данный режим редко оказывается полезным и используется в основном для блокирования агрегатов, как было показано в разделе «Принудительное увеличение номера версии».

А что по поводу режимов блокировки READ и WRITE?

Это старые режимы блокировки, пришедшие из Java Persistence 1.0, и они не должны больше использоваться. `LockModeType.READ` является эквивалентом режима `OPTIMISTIC`, а `LockModeType.WRITE` – эквивалентом режима `OPTIMISTIC_FORCE_INCREMENT`.

При использовании пессимистических блокировок Hibernate блокирует только записи, которые относятся к состоянию экземпляра сущности. Другими словами, если заблокировать экземпляр `Item`, Hibernate заблокирует соответствующую запись в таблице `ITEM`. Если используется стратегия наследования с соединениями, Hibernate поймет это и заблокирует нужные строки в таблицах суперклассов и подклассов. Это также относится к любому отображению сущности во вторичную таблицу. Поскольку Hibernate блокирует всю запись, любое отношение, внешний ключ которого находится в этой записи, также будет заблокировано: связь `Item#seller` будет заблокирована, если столбец внешнего ключа `SELLER_ID` находится в таблице `ITEM`. Но сам экземпляр `Seller` заблокирован не будет! Коллекции и прочие связи класса `Item`, внешний(е) ключ(и) которых находится(ятся) в других таблицах, также не будут заблокированы.

Используя монопольные блокировки в СУБД, можно столкнуться с неудачным завершением транзакций из-за взаимоблокировок.

Расширение области действия блокировки

JPA 2.0 определяет параметр `PessimisticLockScope.EXTENDED`. Он может быть использован в качестве рекомендации для запроса `javax.persistence.lock.scope`. При наличии этого параметра механизм хранения расширит область заблокированных данных, включив в нее любые данные коллекций и таблицы соединения со связями с заблокированными сущностями. На момент написания книги этот параметр еще не был реализован в Hibernate.

11.2.4. Как избежать взаимоблокировок

В СУБД, реализующих изолированность транзакций с применением монопольных блокировок, могут возникать взаимоблокировки. Рассмотрим следующую единицу, обновляющую два экземпляра сущностей `Item` в определенном порядке:

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item itemOne = em.find(Item.class, ITEM_ONE_ID);
itemOne.setName("First new name");
```

```
Item itemTwo = em.find(Item.class, ITEM_TWO_ID);
itemTwo.setName("Second new name");

tx.commit();
em.close();
```

Выталкивая контекст хранения, Hibernate выполнит две SQL-инструкции UPDATE. Первая заблокирует запись, представляющую первый экземпляр Item, а вторая – представляющую второй экземпляр:

```
update ITEM set ... where ID = 1;  ← Заблокирует запись 1
update ITEM set ... where ID = 2;  ← Пытается заблокировать запись 2
```

Взаимоблокировка может возникнуть (а может и не возникнуть!), если в параллельной транзакции будет выполняться похожая процедура, но с обратным порядком изменения экземпляров Item:

```
update ITEM set ... where ID = 2;  ← Заблокирует запись 1
update ITEM set ... where ID = 1;  ← Пытается заблокировать запись 2
```

При возникновении взаимоблокировки обе транзакции оказываются заблокированными – они не могут выполняться дальше, ожидая снятия блокировки. Вероятность взаимоблокировки обычно мала, но в ситуациях с высокой вероятностью параллельного доступа два приложения Hibernate могут выполнить подобное чередующееся изменение. Обратите внимание, что во время тестирования можно не столкнуться со взаимоблокировками (если, конечно, не написать правильных тестов). Но когда приложение работает с высокой транзакционной нагрузкой, взаимоблокировки могут возникать совершенно неожиданно. Обычно по истечении периода ожидания СУБД завершает одну из заблокированных транзакций как неудачную; остальные транзакции могут продолжить выполнение. С другой стороны, СУБД может автоматически определять состояние взаимоблокировки и немедленно прерывать выполнение одной из транзакций, но это зависит от СУБД.

Старайтесь избегать неудачного завершения транзакций, поскольку после них довольно трудно восстанавливать нормальную работу приложения. Одно из решений заключается в использовании для конкретного соединения с базой данных уровня изолированности, обеспечивающего *упорядоченность*, блокирующую всю таблицу при изменении одной записи. Параллельная транзакция должна ожидать, пока первая транзакция завершит работу. С другой стороны, первая транзакция может получить монопольную блокировку всех данных с помощью инструкции SELECT, как было показано в предыдущем разделе. Тогда любая параллельная транзакция также должна будет ждать снятия этих блокировок.

В качестве альтернативы можно использовать программную оптимизацию, которая значительно снижает вероятность взаимоблокировок, упорядочивая выражения UPDATE по значению первичного ключа: Hibernate всегда обновляет запись с первичным ключом 1 перед изменением записи 2, независимо от того, в каком порядке данные были загружены и изменены в приложении. Вы можете использовать эту оптимизацию для всей единицы хранения, добавив параметр hiber-

nate.order_updates в конфигурацию. После этого Hibernate будет упорядочивать все инструкции UPDATE в порядке возрастания значений первичного ключа экземпляров сущностей и элементов коллекций, обнаруженных во время выталкивания контекста (как уже говорилось ранее, вы должны всецело понимать работу транзакций и блокировок в вашей СУБД. Большинство транзакционных гарантий Hibernate наследует от СУБД; например, база данных, поддерживающая версионирование (MVCC), может не блокировать параллельного доступа для чтения, но применять монопольные блокировки для изоляции пишущих транзакций, в результате вы можете столкнуться с взаимоблокировками).

У нас еще не было возможности представить метод EntityManager#lock(). Он принимает загруженный хранимый экземпляр сущности и значение режима блокировки. Устанавливает такие же блокировки, как метод find() и интерфейс Query, с той лишь разницей, что он не загружает экземпляра. Кроме того, если сущность с версионированием блокируется пессимистически, метод lock() сразу же выполняет проверку версии в базе данных и потенциально может возбудить OptimisticLockException. Если представление экземпляра было удалено из базы данных, Hibernate возбудит EntityNotFoundException. Наконец, метод EntityManager#refresh() тоже принимает значение режима блокировки с такой же семантикой.

Мы рассмотрели управление параллельным доступом на самом низком уровне, в базе данных, а также особенности оптимистических и пессимистических блокировок в JPA. Но нужно обсудить еще один аспект параллельного доступа: работу с данными вне транзакции.

11.3. Доступ к данным вне транзакции

Экземпляр JDBC Connection по умолчанию работает в режиме *автоматического подтверждения* (auto-commit). Этот режим удобен для выполнения произвольных запросов SQL.

Представьте, что вы подключились к базе данных с помощью консоли SQL и выполнили несколько запросов, возможно, обновив и удалив несколько записей. Такое интерактивное обращение к данным является произвольным; в большинстве случаев у вас нет плана или последовательности операций, которые можно было бы считать единицей работы. Режим автоматического подтверждения отлично подходит для подобной работы с данными. В конце концов, мало кому захочется каждый раз вводить begin transaction и end transaction для каждого выражения SQL. В режиме автоматического подтверждения каждая (непродолжительная) транзакция начинается и заканчивается вместе с каждым запросом SQL, посылаемым в базу данных. Фактически вы работаете в нетранзакционном режиме, поскольку сеанс в консоли SQL не гарантирует ни атомарности, ни изолированности. (Единственная гарантия – каждое выражение SQL в отдельности будет выполнено атомарно.)

Приложение по определению всегда выполняет запланированную последовательность выражений. По этой причине кажется разумным всегда определять гра-

ницы транзакций, группируя выражения в атомарные единицы, изолированные друг от друга. Тем не менее в JPA с режимом автоматической фиксации связана особая функциональность, и он может понадобиться для реализации длительных диалоговых взаимодействий. Режим автоматического подтверждения в приложениях можно использовать, например, для чтения данных.

11.3.1. Чтение данных в режиме автоматического подтверждения

Рассмотрим следующий пример, сначала загружающий экземпляр `Item`, изменяющий его свойство `name`, а потом откатывающий изменения повторным чтением данных.

Листинг 11.8 ❖ Чтение данных в режиме автоматического подтверждения

Файл: /examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java

```
EntityManager em = JPA.createEntityManager();  ← ❶ Рассинхронизированный режим

Item item = em.find(Item.class, ITEM_ID);       ← ❷ Обращение к базе данных
item.setName("New Name");

assertEquals(  ← ❸ Возвращает начальное значение
    em.createQuery("select i.name from Item i where i.id = :id)")
        .setParameter("id", ITEM_ID).getSingleResult(),
    "Original Name"
);

assertEquals(  ← ❹ Возвращает уже загруженный экземпляр
     ((Item) em.createQuery("select i from Item i where i.id = :id)")
        .setParameter("id", ITEM_ID).getSingleResult()).getName(),
    "New Name"
);

// em.flush();      ← ❺ Возбуждает исключение

em.refresh(item);  ← ❻ Откатывает изменение
assertEquals(item.getName(), "Original Name");

em.close();
```

❶ В момент создания объекта `EntityManager` ни одна транзакция не активна. Контекст хранения находится в особом, *рассинхронизированном* режиме – Hibernate не будет выталкивать контекста автоматически.

❷ Вы можете читать данные из базы; эта операция выполнит инструкцию `SELECT`, передав ее в базу данных в режиме автоматического подтверждения.

❸ Обычно Hibernate выталкивает контекст хранения при выполнении запроса (`Query`). Но поскольку контекст рассинхронизирован, выталкивание не происходит, и запрос возвращает старое, начальное значение. Запросы со скалярными результатами не повторимы: вы видите только те значения, которые есть в базе данных и были переданы в Hibernate в виде объекта `ResultSet`. Кроме того, в синхронизированном режиме вы получите неповторимое чтение.

❹ Получение управляемого экземпляра сущности включает в себя поиск в текущем контексте хранения. Контекст возвращает уже загруженный экземпляр `Item` с измененным названием; значения в базе данных игнорируются. Это чтение экземпляра сущности повторимо, даже в отсутствие системной транзакции.
❺ При попытке вытолкнуть контекст хранения вручную, чтобы сохранить новое значение `Item#name`, Hibernate возбудит `javax.persistence.TransactionRequiredException`. Нельзя выполнять операцию `UPDATE` в рассинхронизированном режиме, потому что он не позволяет откатить изменения.
❻ Откатить изменения можно вызовом метода `refresh()`. Он загружает текущее состояние экземпляра `Item` из базы данных и перезаписывает сделанные в памяти изменения.

Используя рассинхронизированный контекст хранения, данные можно читать в режиме автоматического подтверждения, используя запросы или методы `find()`, `getReference()` и `refresh()`. Также можно загружать данные при необходимости: инициализация прокси-объектов происходит при обращении к ним, а коллекции загружаются в момент начала обхода элементов. Но если попытаться вытолкнуть контекст хранения или заблокировать данные в режиме, отличном от `LockModeType.NONE`, возникнет исключение `TransactionRequiredException`.

Пока что режим автоматического подтверждения кажется не особо полезным. Действительно, многие разработчики заблуждаются в мотивах применения режима автоматического подтверждения:

- множество коротких транзакций для отдельных выражений (суть режима автоматического подтверждения) не улучшит производительности приложения;
- ухудшается масштабируемость приложения: одна продолжительная транзакция вместо нескольких коротких для отдельных выражений SQL может дольше удерживать блокировку. Но эта проблема незначительна, поскольку Hibernate выполняет запись в базу данных как можно позже, ближе к концу транзакции (выталкивание контекста происходит во время подтверждения), поэтому фактически база данных удерживает блокировку на запись лишь короткий промежуток времени;
- слабые гарантии изолированности, если приложение будет менять данные параллельно. Повторимое чтение с помощью блокировок на запись невозможно в режиме автоматического подтверждения (естественно, здесь на помощь приходит кэш контекста хранения);
- если СУБД реализует управление параллельным доступом с помощью версионирования (MVCC), как, например, Oracle, PostreSQL, Informix или Firebird, у вас наверняка появится желание использовать возможность *изоляции моментальных снимков*, чтобы избежать неповторимого и фантомного чтения. Каждая транзакция получает собственный моментальный снимок данных; вы видите ту версию данных (внутри базы данных), какой она была перед началом транзакции. При работе в режиме автоматического подтверждения изоляция моментального снимка не имеет смысла, поскольку отсутствует область действия транзакции;

- снижается ясность программного кода. Теперь каждый, кто читает ваш код, должен обращать особое внимание, присоединен ли контекст хранения к транзакции или находится в рассинхронизированном режиме. Если вы постоянно группируете операции внутри системной транзакции даже при чтении данных, каждый сможет следовать этому простому правилу, что позволит снизить количество трудных для обнаружения проблем с многопоточностью.

Какими же преимуществами обладает рассинхронизированный контекст хранения? Если выталкивание контекста не происходит автоматически, вы можете подготовить все изменения вне транзакции, поместив их в *очередь*.

11.3.2. Создание очереди изменений

Следующий пример демонстрирует сохранение экземпляра `Item` с помощью рассинхронизированного экземпляра `EntityManager`:

Файл: /examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java

```
EntityManager em = JPA.createEntityManager();

Item newItem = new Item("New Item");
em.persist(newItem)   ← ❶ Сохранение временного экземпляра
assertNotNull(newItem.getId());

tx.begin();    ← ❷ Сохранение изменений
if (!em.isJoinedToTransaction())
    em.joinTransaction();
tx.commit();  ← Выталкивание!
em.close();
```

❶ Сохранить временный экземпляр сущности в рассинхронизированном контексте можно вызовом метода `persist()`. Hibernate просто получит новое значение идентификатора, обратившись к последовательности в базе данных, и назначит его экземпляру. Теперь экземпляр находится в хранимом состоянии внутри контекста, но SQL-инструкция `INSERT` еще не выполнилась. Обратите внимание, что такое возможно только при использовании генератора идентификаторов, срабатывающего *перед вставкой*; см. раздел 4.2.5.

❷ Когда изменения будут готовы к сохранению, присоедините контекст к транзакции. Синхронизация и выталкивание контекста произойдут как обычно, во время подтверждения транзакции. Hibernate запишет все накопленные изменения в базу данных.

Также в очередь можно поместить операцию слияния для отсоединенного экземпляра сущности:

Файл: /examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java

```
detachedItem.setName("New Name");
EntityManager em = JPA.createEntityManager();
```

```
Item mergedItem = em.merge(detachedItem);

tx.begin();
em.joinTransaction();
tx.commit();  ← Выталкивание контекста!
em.close();
```

Hibernate выполнит инструкцию SELECT в режиме автоматического подтверждения во время вызова merge(). Но отложит выполнение UPDATE до подтверждения присоединенной транзакции.

Очередь можно создать и для удаления экземпляров сущностей и операций DELETE:

Файл: /examples/src/test/java/org/jpwh/test/concurrency/NonTransactional.java

```
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
em.remove(item);

tx.begin();
em.joinTransaction();
tx.commit();  ← Выталкивание контекста!
em.close();
```

Рассинхронизированный контекст позволяет отделять операции с хранимыми сущностями от транзакций. Эта особенность поведения EntityManager сыграет важную роль в дальнейшем, когда мы приступим к обсуждению архитектуры приложения. Возможность накапливать изменения данных независимо от транзакций (а также запросов клиент/сервер) является главной особенностью контекста хранения.

Режим выталкивания контекста вручную

В Hibernate имеется метод Session#setFlushMode() с дополнительным параметром FlushMode.MANUAL, отключающим автоматическое выталкивание контекста хранения даже в случае подтверждения присоединенной транзакции. В этом режиме вам придется самостоятельно вызывать flush() для синхронизации с базой данных. Спецификация JPA исповедует идею: «подтверждение транзакции всегда должно приводить к записи любых изменений». Поэтому в ней чтение было отделено от записи с помощью *рассинхронизированного* режима. Если вы не согласны с этим и/или не хотите автоматически подтверждать изменения после выполнения инструкций, можете использовать режим ручного сохранения контекста с помощью Session API. В результате вы получите обычные границы транзакций для всех единиц работы, повторимое чтение и даже изоляцию моментальных снимков (если поддерживаются) и сможете накапливать изменения в контексте хранения для последующей обработки и ручного вызова метода flush() перед подтверждением транзакции.

11.4. Резюме

- Вы узнали, как использовать транзакции, многопоточность, изолированность и блокировки.
- Hibernate полагается на механизм управления параллельным доступом в базе данных, но дает лучшие гарантии изолированности транзакций благодаря автоматическому версионированию и кэшу контекста хранения.
- Мы обсудили программное определение границ транзакций и обработку исключений.
- Вы познакомились с оптимистическим управлением параллельным доступом и явными пессимистическими блокировками.
- Вы узнали, как работать с режимом автоматического подтверждения и рассинхронизированным контекстом хранения, а также как накапливать изменения.

Глава 12

Планы извлечения, стратегии и профили

В этой главе:
- отложенная и немедленная загрузка;
- планы извлечения, стратегии и профили;
- оптимизация выполнения SQL.

В этой главе мы исследуем решение фундаментальной проблемы ORM – обхода графа объектов, о чем упоминалось в разделе 1.2.5. Мы покажем, как извлекать информацию из базы данных и как этот процесс можно оптимизировать.

Hibernate поддерживает следующие способы загрузки информации из базы данных в память.

- Если известно уникальное значение идентификатора экземпляра сущности, самый удобный способ получить его – выполнить поиск по идентификатору, например `entityManager.find(Item.class, 123)`.
- Имеется возможность начать обход графа объектов с уже загруженного экземпляра сущности, обращаясь к связанным сущностям через методы доступа к свойствам, такие как `someItem.getSeller().getAddress().getCity()` и т. д. Элементы отображаемых коллекций также могут загружаться по требованию в начале обхода коллекции. Hibernate автоматически загружает вершины графа объектов, если контекст хранения еще открыт. Эта глава расскажет, какие данные загружаются при вызове методов доступа и обходе коллекций и каким способом происходит загрузка.
- Поддерживается полноценный объектно-ориентированный язык запросов Java Persistence Query Language (JPQL), использующий строковое представление, например `select i from Item i where i.id = ?`.
- Интерфейс `CriteriaQuery` реализует типобезопасный и объектно-ориентированный способ выполнения запросов без использования строк.
- Имеется возможность писать запросы на чистом SQL, вызывать хранимые процедуры и переложить на плечи фреймворка Hibernate отображение результатов запроса JDBC в экземпляры классов предметной модели.

В своих приложениях JPA вы будете применять комбинацию этих подходов, но в этой главе мы не станем подробно разбирать каждый способ извлечения данных. Вы уже достаточно хорошо знакомы с основами Java Persistence API, чтобы самостоятельно реализовать загрузку по идентификатору. Мы постарались сделать примеры с JPQL и `CriteriaQuery` как можно более простыми, чтобы вам не пришлось использовать средства отображения запросов SQL. Поскольку эта функциональность довольно запутанна, мы исследуем ее позже в главах 15 и 17.

Главные нововведения в JPA 2

- Теперь можно вручную проверять состояние загрузки сущности или ее свойств, используя статические методы вспомогательного класса `PersistenceUtil`.
- Можно создавать стандартизованные декларативные планы извлечения с помощью нового интерфейса `EntityGraph`.

Эта глава рассказывает, что происходит за кулисами, когда выполняется обход графа объектов предметной модели и когда Hibernate загружает данные по требованию. Во всех примерах мы будем показывать в комментариях код SQL, выполняемый фреймворком Hibernate.

Что именно загрузит Hibernate, зависит от *плана извлечения*: вы определяете (под)граф множества объектов для загрузки. Затем выбираете *стратегию извлечения*, определяя, *как* должны загружаться данные. План и стратегию можно сохранить в виде *профиля извлечения*, чтобы использовать их повторно.

Определение плана извлечения, *какие* данные должен загрузить Hibernate, зависит от двух фундаментальных способов загрузки вершин графа объектов: *отложенного* (lazy) и *немедленного* (eager).

12.1. Отложенная и немедленная загрузка

В какой-то момент приходится выбирать, какие данные должны быть загружены в память из базы. Что окажется в памяти и будет загружено в контекст хранения при вызове `entityManager.find(Item.class, 123)`? Что произойдет, если вместо этого вызвать `EntityManager#getReference()`?

В отображении предметной модели для связей и коллекций можно определить глобальный *план извлечения по умолчанию* с помощью параметров `FetchType.LAZY` и `FetchType.EAGER`. Он будет использоваться по умолчанию для всех операций с участием классов предметной модели. Он всегда будет использоваться при загрузке экземпляра сущности по идентификатору, обходе графа объектов по связям и в итерациях по хранимым коллекциям.

В качестве плана извлечения по умолчанию мы рекомендуем использовать стратегию *отложенной* загрузки для всех сущностей и коллекций. Если все связи и коллекции отобразить с параметром `FetchType.LAZY`, Hibernate будет загружать только данные, к которым происходит обращение в данный момент. Во время обхода графа экземпляров предметной модели Hibernate будет загружать данные по

требованию, шаг за шагом. При необходимости это поведение можно переопределять в отдельных ситуациях.

Для реализации отложенной загрузки Hibernate использует сгенерированные во время выполнения заглушки, называемые *прокси-объектами* или, в случае коллекций, *умными обертками*.

12.1.1. Прокси-объекты

Рассмотрим метод `getReference()` интерфейса `EntityManager`. Мы впервые познакомились с этим интерфейсом в разделе 10.2.3 и узнали, как он может возвращать прокси-объекты. Давайте исследуем эту важную особенность далее и узнаем, как же работают прокси-объекты:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Item item = em.getReference(Item.class, ITEM_ID);  ← Никакого SELECT

assertEquals(item.getId(), ITEM_ID);  ← Вызов метода чтения идентификатора (без доступа
                                        к членам класса!) не вызовет инициализацию
```

Этот код не повлечет отправки SQL-запроса в базу данных. Hibernate просто создаст прокси-объект `Item`: он выглядит (и кажется) как настоящий объект, но это всего лишь заглушка. В контексте хранения, в памяти, появится прокси-объект в хранимом состоянии, как показано на рис. 12.1.

Рис. 12.1 ❖ Контекст хранения содержит прокси-объект `Item`

Прокси-объект – это экземпляр подкласса `Item`, сгенерированный во время выполнения и хранящий значение идентификатора представляемого им экземпляра сущности. Вот почему Hibernate (наряду с JPA) требует, чтобы класс сущности имел общедоступный или защищенный конструктор без аргументов (класс также может иметь другие конструкторы). Чтобы Hibernate смог сгенерировать прокси-объект, класс сущности и его методы не должны быть финальными (final). Обратите внимание, что спецификация JPA вообще не упоминает прокси-объектов; способ осуществления отложенной загрузки полностью зависит от реализации JPA.

Если вызвать любой метод прокси-объекта, отличный от метода чтения идентификатора, произойдут обращение к базе данных и инициализация прокси-объекта. При вызове `item.getName()` выполнится SQL-инструкция `SELECT`, которая

загрузит экземпляр `Item`. Вызов `item.getId()` в предыдущем примере не повлек инициализации, поскольку в данном отображении `getId()` *является* методом чтения идентификатора; метод `getId()` был отмечен аннотацией `@Id`. Если поместить аннотацию `@Id` перед полем, вызов `getId()`, как и любого другого метода, повлек бы инициализацию прокси-объекта! (Как вы наверняка помните, мы обычно рекомендуем применять аннотации отображения и доступа к полям класса, поскольку это дает больше свободы при проектировании методов доступа; см. раздел 3.2.3. Только вам решать, является ли более важным вызов `getId()` без инициализации прокси-объекта.)

Будьте осторожней, сравнивая классы при работе с прокси-объектами. Поскольку прокси-класс генерируется фреймворком Hibernate, он получает довольно забавное имя, не совпадающее со значением `Item.class`:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
assertNotEquals(item.getClass(), Item.class);

assertEquals(
    HibernateProxyHelper.getClassWithoutInitializingProxy(item),
    Item.class
);
```

Класс сгенерирован во время выполнения и получает примерно такое имя: Item_$$_javassist_1

Если потребуется получить действительный тип, представляемый прокси-классом, используйте класс `HibernateProxyHelper`.

В JPA имеется класс `PersistenceUtil`, который можно использовать для проверки состояния инициализации сущности или любого ее атрибута:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertFalse(persistenceUtil.isLoaded(item));
assertFalse(persistenceUtil.isLoaded(item, "seller"));

assertFalse(Hibernate.isInitialized(item));
// assertFalse(Hibernate.isInitialized(item.getSeller()));
```

Вызовет инициализацию item!

Статический метод `isLoaded()` также принимает имя свойства данного экземпляра сущности (прокси-объекта) и проверяет его состояние инициализации. Hibernate поддерживает альтернативный метод `Hibernate.isInitialized()`. Но если вызвать `item.getSeller()`, сначала произойдет инициализация прокси-объекта `item`!

Особенности Hibernate

В Hibernate также имеется метод для инициализации прокси-объектов по требованию:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Hibernate.initialize(item);
// select * from ITEM where ID = ?
```

```
assertFalse(Hibernate.isInitialized(item.getSeller()));

Hibernate.initialize(item.getSeller());
// select * from USERS where ID = ?
```

Убедитесь, что значение по умолчанию EAGER в аннотации @ManyToOne было заменено на LAZY

Первый вызов произведет обращение к базе данных и загрузит информацию для экземпляра `Item`, инициализируя такие свойства прокси-объекта, как имя, цена и т. д.

Свойство `seller` объекта `Item` представляет связь `@ManyToOne`, отображаемую с параметром `FetchType.LAZY`, поэтому во время загрузки экземпляра `Item` фреймворк Hibernate создаст прокси-объект `User`. Вы можете проверить состояние прокси-объекта `seller` и загрузить его вручную точно так же, как экземпляр `Item`. Помните, что в JPA аннотация `@ManyToOne` по умолчанию получает параметр `FetchType.EAGER`! Обычно его желательно переопределить, чтобы использовать план отложенного извлечения, как впервые было показано в разделе 7.3.1, а теперь и здесь:

Файл: /model/src/main/java/org/jpwh/model/fetching/proxy/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    public User getSeller() {
        return seller;
    }
    // ...
}
```

Используя план отложенного извлечения, можно столкнуться с исключением `LazyInitializationException`. Рассмотрим следующий код:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?
```

❶ Загрузка экземпляра Item

```
em.detach(item);
em.detach(item.getSeller());
// em.close();
```

❷ Отсоединение данных

❸ Вспомогательный класс PersistenceUtil

```
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertTrue(persistenceUtil.isLoaded(item));

assertFalse(persistenceUtil.isLoaded(item, "seller"));

assertEquals(item.getSeller().getId(), USER_ID);
// assertNotNull(item.getSeller().getUsername());
```

❹ Вызывает метод чтения

Возбудит исключение!

❶ Экземпляр сущности `Item` загружается в контекст хранения. Свойство `seller` не инициализируется, так как это прокси-объект `User`.

❷ Вы можете отсоединить данные от контекста хранения вручную или закрыть контекст, отсоединив все.

❸ Статический вспомогательный класс `PersistenceUtil` работает без контекста хранения. В любой момент вы можете проверить, были ли загружены данные, к которым предполагается обратиться.

❹ В отсоединенном состоянии допускается вызывать метод чтения идентификатора прокси-объекта `User`. Но вызов любого другого метода прокси-объекта, такого как `getUsername()`, возбудит исключение `LazyInitializationException`. Данные могут загружаться по требованию, только пока прокси-объект находится под управлением контекста хранения, но не в отсоединенном состоянии.

Как работает отложенная загрузка связей *один к одному*?

Отложенная загрузка связи *один к одному* иногда вводит новых пользователей Hibernate в замешательство. Например, связь *один к одному*, использующая разделяемый первичный ключ (см. раздел 8.1.1), может быть представлена прокси-объектом только при использовании параметра `optional=false`. Например, адрес (`Address`) всегда имеет ссылку на пользователя (`User`). Если связь может иметь значение null и не является обязательной, Hibernate должен сначала обратиться к базе данных и выяснить, нужно ли возвращать прокси-объект или null; но цель отложенной загрузки как раз в том и состоит, чтобы избежать обращений к базе данных. Вы всегда можете настроить отложенную загрузку необязательных связей *один к одному* путем перехвата вызовов, и мы обсудим этот прием ниже в данной главе.

Прокси-объекты Hibernate могут пригодиться не только в случае простой отложенной загрузки. Например, можно сохранить новую ставку (`Bid`), не загружая в память никаких данных:

```
Item item = em.getReference(Item.class, ITEM_ID);
User user = em.getReference(User.class, USER_ID);

Bid newBid = new Bid(new BigDecimal("99.00"));
newBid.setItem(item);
newBid.setBidder(user);

em.persist(newBid);
// insert into BID values (?, ? ,? , ...)
```

В этой процедуре не выполняется ни одного SQL-запроса SELECT, только одна инструкция INSERT

Два первых вызова создадут прокси-объекты товара (`Item`) и пользователя (`User`) соответственно. Затем полям связей временного экземпляра ставки (`Bid`) с товаром (`item`) и пользователем (`bidder`) будут присвоены значения в виде прокси-объектов. Вызов `persist()` поместит в очередь одну SQL-инструкцию `INSERT`, и для создания записи в таблице `BID` не понадобится ни одной инструкции `SELECT`, потому что значения всех (внешних) ключей доступны как значения идентификаторов прокси-объектов `Item` и `User`.

Динамическое создание прокси-классов фреймворком Hibernate во время выполнения – превосходное решение для прозрачной отложенной загрузки. Классы предметной модели не должны наследовать каких-либо специальных (супер)типов, как это было в более старых решениях ORM. Также не требуется генериро-

вать дополнительный код или выполнять постобработку байт-кода, что упрощает процедуру сборки проекта. Но вы должны иметь в виду некоторые негативные аспекты:

- когда полиморфные ассоциации проверяются оператором `instanceof`, сгенерированные во время выполнения прокси-классы не являются полностью прозрачным решением, как было показано в разделе 6.8.1;
- при работе с прокси-классами сущностей нужно избегать прямых обращений к полям в реализациях методов `equals()` и `hashCode()`, как обсуждалось в разделе 10.3.2;
- прокси-объекты могут использоваться только для отложенной загрузки связей сущностей. Они не могут применяться для отложенной загрузки полей простых типов или встроенных компонентов, таких как `Item#description` или `User#homeAddress`. Если для такого поля указать подсказку `@Basic(fetch = FetchType.LAZY)`, Hibernate ее проигнорирует; значение будет загружено сразу же при загрузке экземпляра сущности-владельца. Отложенную загрузку можно реализовать перехватом вызовов, но мы считаем, что подобная оптимизация редко бывает полезной. Оптимизация на уровне отдельных столбцов, выбираемых запросом SQL, не нужна, если только вы не работаете (а) с большим количеством необязательных столбцов (или столбцов, которые могут хранить null) или (б) со столбцами, содержащими большие объемы данных, которые нужно загружать по требованию из-за физических ограничений системы. Вместо этого объемные значения лучше представлять с помощью объектов логических указателей (LOB); они изначально поддерживают отложенную загрузку (см. раздел «Двоичные типы и типы для представления больших значений» в главе 5).

Прокси-объекты обеспечивают отложенную загрузку экземпляров сущностей. Для хранимых коллекций Hibernate применяет несколько иной подход.

12.1.2. Отложенная загрузка хранимых коллекций

Хранимые коллекции отображаются либо с помощью аннотации `@ElementCollection` (коллекции элементов простых или встраиваемых типов), либо с помощью аннотаций `@OneToMany` и `@ManyToMany` (связи между сущностями). К таким коллекциям, в отличие от `@ManyToOne`, отложенная загрузка применяется по умолчанию. То есть в отображении можно не указывать параметра `FetchType.LAZY`.

При загрузке товара (`Item`) Hibernate не станет тут же загружать коллекцию изображений (`images`). Коллекция `bids` (ставки), представляющая отношение *один ко многим*, также будет загружена по требованию, когда произойдет обращение к ней:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?

Set<Bid> bids = item.getBids();  ← Коллекция не инициализирована
```

```
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertFalse(persistenceUtil.isLoaded(item, "bids"));

assertTrue(Set.class.isAssignableFrom(bids.getClass()));  ← Это множество Set

assertNotEquals(bids.getClass(), HashSet.class);  ← Это не класс HashSet
assertEquals(bids.getClass(),
    org.hibernate.collection.internal.PersistentSet.class);
```

Операция `find()` загрузит экземпляр сущности `Item` в контекст хранения, как видно на рис. 12.2. Экземпляр `Item` имеет неинициализированную ссылку `seller` на прокси-объект `User`. Также он имеет ссылку на неинициализированное множество `Set` ставок (`bids`) и неинициализированный список `List` изображений `images`.

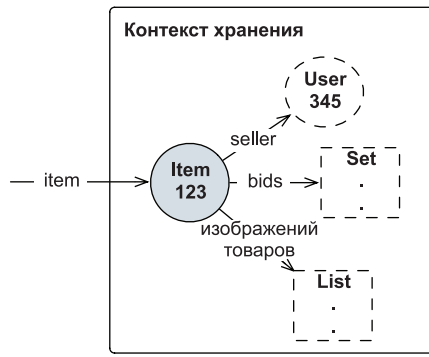


Рис. 12.2 ❖ Прокси-объекты и обертки коллекций представляют границы загруженного графа сущностей

Hibernate реализует отложенную загрузку (и проверку состояния объектов) коллекций с помощью собственного механизма *оберток коллекций*. Хотя коллекция `bids` кажется похожей на `Set`, Hibernate поменял ее реализацию на `org.hibernate.collection.internal.PersistentSet`.

Это не `HashSet`, но имеет схожее поведение. Вот почему в предметной модели так важно программировать с использованием интерфейсов и полагаться только на `Set`, а не на `HashSet`. Списки и словари работают аналогично.

Эти специальные коллекции могут обнаруживать обращение к ним и загружать данные в этот момент. Как только вы начнете обход множества `bids`, сразу же будет загружена коллекция вместе со ставками:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyProxyCollections.java

```
Bid firstBid = bids.iterator().next();
// select * from BID where ITEM_ID = ?

// Альтернативный вариант: Hibernate.initialize(bids);
```

Для прокси-классов сущностей существует альтернативный вариант – статический вспомогательный метод `Hibernate.initialize()`, загружающий коллекцию. Он загрузит всю коллекцию; вы не сможете, к примеру, загрузить только две первые ставки. Для реализации подобного поведения придется написать запрос.

Особенности Hibernate

Чтобы избавить от необходимости писать множество тривиальных запросов, Hibernate предоставляет свой способ отображения коллекций:

Файл: /model/src/main/java/org/jpwh/model/fetching/proxy/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.LazyCollection(
        org.hibernate.annotations.LazyCollectionOption.EXTRA
    )
    public Set<Bid> getBids() {
        return bids;
    }
    // ...
}
```

Параметр `LazyCollectionOption.EXTRA` включает поддежку операций с коллекцией, не вызывающих ее инициализацию. Например, можно узнать размер коллекции:

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?

assertEquals(item.getBids().size(), 3);
// select count(b) from BID b where b.ITEM_ID = ?
```

Операция `size()` выполнит SQL-запрос `SELECT COUNT()`, но не загрузит коллекцию `bids` в память. Для коллекций с дополнительным параметром подобные запросы будут выполняться также для операций `isEmpty()` и `contains()`. При вызове метода `add()` множество `Set` с дополнительным параметром проверит повторяющиеся элементы с помощью простого запроса. Список `List` с дополнительным параметром загрузит только один элемент при вызове `get(index)`. Словарь `Map` получит дополнительные отложенные операции: `containsKey()` и `containsValue()`.

Прокси-объекты и умные коллекции в Hibernate являются лишь одной из возможных реализаций отложенной загрузки, имеющих хорошее соотношение между стоимостью и функциональностью. Альтернативой, о которой упоминалось ранее, является *перехват вызовов*.

Особенности Hibernate

12.1.3. Реализация отложенной загрузки путем перехвата вызовов

Фундаментальная проблема отложенной загрузки состоит в том, что реализация JPA должна знать, в какой момент загружать поле `seller` объекта `Item` или коллекцию `bids`. Вместо создания прокси-классов во время выполнения и работы с умными коллекциями другие реализации JPA полагаются исключительно на *перехват* вызовов методов. Например, при вызове `someItem.getSeller()` реализация JPA *перехватит* этот вызов и загрузит экземпляр `User`, представляющий продавца товара (поле `seller`).

Такой подход требует специального кода в классе `Item` для реализации перехвата: метод `getSeller()` или поле `seller` должно быть обернуто. Поскольку мало у кого возникает желание писать подобный код вручную, как правило, после компиляции классов предметной модели запускается *оптимизатор байт-кода* (поставляемый вместе с реализацией JPA). Он вставляет необходимый код перехвата вызовов в скомпилированные классы, взаимодействуя с полями и методами на уровне байт-кода.

Давайте обсудим реализацию отложенной загрузки путем перехвата на паре примеров. Сначала отключим генерацию прокси-классов в Hibernate:

Файл: /model/src/main/java/org/jpwh/model/fetching/interception/User.java

```
@Entity
@org.hibernate.annotations.Proxy(lazy = false)
public class User {
    // ...
}
```

Hibernate больше не будет генерировать прокси-класс для сущности `User`. Теперь при вызове `entityManager.getReference(User.class, USER_ID)` будет выполнена инструкция `SELECT`, так же как при вызове `find()`:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyInterception.java

```
User user = em.getReference(User.class, USER_ID);
// select * from USERS where ID = ?

assertTrue(Hibernate.isInitialized(user));
```

Прокси-классы не используются. getReference() вернет инициализированный экземпляр

Для связей сущностей, направленных к классу `User`, таких как поле `seller` класса `Item`, подсказка `FetchType.LAZY` не будет работать:

Файл: /model/src/main/java/org/jpwh/model/fetching/interception/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    @org.hibernate.annotations.LazyToOne(
```

Никакого эффекта – отсутствует прокси-класс User

Требует внедрения дополнительного байт-кода!

```
        org.hibernate.annotations.LazyToOneOption.NO_PROXY
    )
    protected User seller;

    // ...
}
```

Вместо этого настройка `LazyToOneOption.NO_PROXY` сообщит Hibernate, что оптимизатор байт-кода добавит код перехвата обращений к свойству `seller`. Если не использовать этот параметр или не запускать оптимизатора байт-кода, эта связь будет загружаться и заполняться данными вместе с экземпляром `Item`, поскольку создание прокси-объектов для класса `User` отключено.

Если запустить оптимизатор байт-кода, Hibernate будет перехватывать обращения к полю `seller` и вызывать загрузку при обращении к нему:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyInterception.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?

assertEquals(item.getSeller().getId(), USER_ID);
// select * from USERS where ID = ?   ← Даже вызов item.getSeller() повлечет выполнение SELECT
```

Это поведение отличается от поведения прокси-объектов. Как вы помните, мы могли вызвать метод `User#getId()` прокси-объекта без инициализации экземпляра. При использовании перехвата вызовов любое обращение к полю `seller` и вызов `getSeller()` повлекут инициализацию.

Для связей сущностей обычно лучше использовать прокси-объекты. Более распространенный вариант использования перехватчиков – работа со свойствами простых типов, таких как `String` или `byte[]`, возможно, имеющих большой объем. Мы могли бы утверждать, что для больших строк и двоичных данных лучше использовать объекты логических указателей (LOB), как показано в разделе «Двоичные типы и типы для представления больших значений», но вы, возможно, не захотите добавлять типов `java.sql.Blob` или `java.sql.Clob` в свою предметную модель. Прием перехвата вызовов позволяет загружать простые поля типа `String` или `byte[]` по требованию:

Файл: /model/src/main/java/org/jpwh/model/fetching/interception/Item.java

```
@Entity
public class Item {

    @Basic(fetch = FetchType.LAZY)
    protected String description;

    // ...
}
```

Если обработать скомпилированные классы оптимизатором байт-кода, для поля `Item#description` будет использована отложенная загрузка. Если не запускать оптимизатора байт-кода, например во время разработки, свойство типа `String` будет загружаться вместе с экземпляром `Item`.

При использовании перехвата вызовов обратите внимание, что Hibernate загрузит *все* свойства сущности или встраиваемого класса с настроенной отложенной загрузкой, даже если должно быть загружено только одно:

Файл: /examples/src/test/java/org/jpwh/test/fetching/LazyInterception.java

```
Item item = em.find(Item.class, ITEM_ID);
// select NAME, AUCTIONEND, ... from ITEM where ID = ?

assertTrue(item.getDescription().length() > 0);
// select DESCRIPTION from ITEM where ID = ?
// select * from USERS where ID = ?
```

Обращение к одному свойству загрузит все остальные (описание, продавец и пр.)

При загрузке описания (`description`) товара (`Item`) Hibernate тут же загрузит продавца (`seller`) и все остальные поля, доступ к которым перехватывается. На момент написания книги в Hibernate еще не было групп загрузки: загружалось все или ничего.

Недостатками перехвата вызовов являются необходимость выполнения оптимизатора байт-кода во время каждой сборки классов предметной модели и ожидание завершения его работы. Вы можете не выполнять оптимизацию во время разработки, если, к примеру, поведение приложения не зависит от состояния загрузки описания товара. Затем во время сборки, тестирования и создания готового программного продукта можно запустить оптимизатор.

Оптимизатор байт-кода в Hibernate 5

К сожалению, мы не можем гарантировать работоспособности приведенных здесь примеров перехвата вызовов в последней версии Hibernate 5. Оптимизатор байт-кода Hibernate 5 был переписан и теперь обеспечивает больше, чем просто перехват вызовов для поддержки отложенной загрузки: он может внедрять в классы предметной модели код, ускоряющий проверку состояния объектов и автоматически управляющий двунаправленными связями сущностей. Тем не менее на момент написания книги мы не смогли заставить этот новый оптимизатор работать, а его разработка еще продолжалась. За более подробной информацией об оптимизаторе и способах его настройки обращайтесь к текущей документации Hibernate в проекте.

Только вам решать, использовать ли прием перехвата вызовов для поддержки отложенной загрузки, но, судя по нашему опыту, подходящих вариантов для его использования довольно мало. Обратите внимание, что при обсуждении перехвата вызовов мы не упомянули об обертках коллекций: несмотря на возможность настроить перехват вызовов для полей, представляющих коллекции, Hibernate все равно будет использовать собственные умные обертки коллекций. Причина в том, что эти обертки, в отличие от прокси-объектов, помимо поддержки отложенной загрузки, нужны также для других целей. Например, Hibernate исполь-

зует их для обнаружения добавления и удаления элементов коллекций во время проверки состояния объектов. Вы не сможете отключить обертки коллекций в отображениях – они используются всегда (конечно, вы не *обязаны* отображать хранимых коллекций; это лишь дополнительная функциональность, а не требование. См. предыдущее обсуждение в разделе 7.1). С другой стороны, для хранимых массивов отложенная загрузка может применяться только путем перехвата вызовов – их нельзя обернуть так же, как коллекции.

Вот вы и познакомились со всеми доступными способами организации отложенной загрузки в Hibernate. Далее мы посмотрим на противоположную сторону – немедленную загрузку данных.

12.1.4. Немедленная загрузка коллекций и ассоциаций

Мы рекомендуем применять по умолчанию план отложенного извлечения, задаваемый параметром `FetchType.LAZY`, ко всем отображениям связей и коллекций. Но иногда, хоть и нечасто, может понадобиться обратное: указать, что конкретная связь или коллекция должна загружаться всегда, чтобы гарантировать наличие данных в памяти без дополнительного обращения к базе данных.

Но еще более важной может оказаться доступность свойства `seller` объекта `Item`, даже когда объект `Item` находится в отсоединенном состоянии. После закрытия контекста хранения отложенная загрузка становится недоступной. Если `seller` указывает на неинициализированный прокси-объект, при попытке обратиться к нему в отсоединенном состоянии вы получите `LazyInitializationException`. Чтобы данные были доступны в отсоединенном состоянии, нужно предварительно загрузить их вручную, пока контекст открыт, или поменять план извлечения с отложенного на немедленный, если требуется, чтобы они загружались *всегда*.

Предположим, что нам требуется всегда загружать свойства `seller` и `bids` объекта `Item`:

Файл: /model/src/main/java/org/jpwh/model/fetching/eagerjoin/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)  ← Значение по умолчанию
    protected User seller;

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)  ← Не рекомендуется
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

В отличие от параметра `FetchType.LAZY`, являющегося лишь рекомендацией, которую реализация JPA может игнорировать, параметр `FetchType.EAGER` – это требование. Реализация должна обеспечить загрузку данных и их доступность в отсоединенном состоянии; она не может игнорировать эту настройку.

Рассмотрим отображение коллекции: насколько оправдано решение загружать все ставки за товар в память при загрузке самого товара? Даже если потребуется только отобразить название товара или узнать, когда заканчивается аукцион, все ставки все равно будут загружаться в память. Немедленная загрузка всех коллекций путем применения параметра `FetchType.EAGER` в качестве плана извлечения по умолчанию в отображениях – не самая лучшая стратегия. Используя немедленную загрузку для нескольких коллекций, вы неизбежно столкнетесь с *проблемой декартова произведения*, которую мы обсудим далее в этой главе. Лучше всего оставить для коллекций значение по умолчанию `FetchType.LAZY`.

Теперь при вызове `find()` для поиска экземпляра `Item` (или при вынужденной инициализации прокси-объекта `Item`) свойства `seller` (продавец) и `bids` (ставки) будут загружены в контекст хранения в виде хранимых экземпляров:

Файл: /examples/src/test/java/org/jpwh/test/fetching/EagerJoin.java

```
Item item = em.find(Item.class, ITEM_ID);
// select i.*, u.*, b.*
//  from ITEM i
//   left outer join USERS u on u.ID = i.SELLER_ID
//   left outer join BID b on b.ITEM_ID = i.ID
// where i.ID = ?

em.detach(item);  ← Извлечение окончено; отложенная загрузка больше не работает

assertEquals(item.getBids().size(), 3);  ← В отсоединенном состоянии доступны предложения цены...
assertNotNull(item.getBids().iterator().next().getAmount());

assertEquals(item.getSeller().getUsername(), "johndoe");  ← ...и продавец
```

Для загрузки данных при вызове `find()` Hibernate выполнит единственное SQL-выражение `SELECT` с предложением `JOIN`, включающим три таблицы. Содержимое контекста хранения для этого случая показано на рис. 12.3. Обратите внимание, как представлены границы загруженного графа объектов: коллекция `images` не была загружена, а каждый экземпляр `Bid` ссылается на неинициализированный прокси-объект `User` с помощью поля `bidder`. Если сейчас отсоединить экземпляр `Item`, сохранится возможность обращаться к загруженным полям `seller` и `bids`, не вызывая исключения `LazyInitializationException`. Но если попытаться обратиться к полю `images` или одному из прокси-объектов через поле `bidder`, будет возбуждено исключение!

В следующих примерах предполагается, что в предметной модели по умолчанию используется план отложенного извлечения. Hibernate будет загружать только те данные, которые запрашиваются явно, и только те связи и коллекции, к которым происходят обращения.

Далее мы обсудим, *как* данные должны загружаться при поиске экземпляра сущности по идентификатору и при обходе графа объектов с помощью указателей на отображаемые связи и коллекции. Нас интересует не только выполняемый код SQL, но и поиск идеальной *стратегии извлечения*.

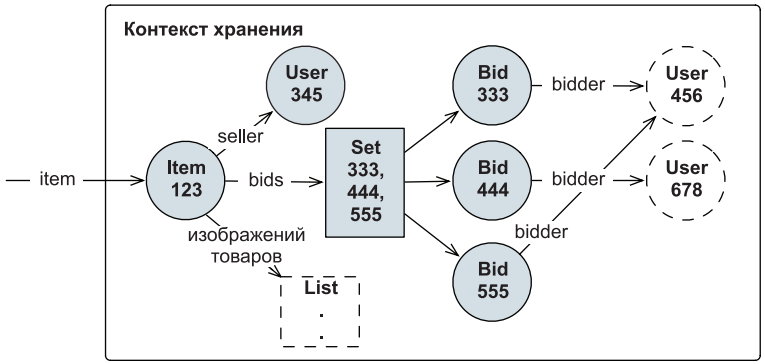


Рис. 12.3 ❖ Загружены продавец и ставки для товара (`Item`)

12.2. Выбор стратегии извлечения

Для загрузки данных в пямять Hibernate выполняет SQL-выражения `SELECT`. При этом в зависимости от числа участвующих таблиц и выбранной *стратегии извлечения* могут выполняться одно или несколько выражений `SELECT`. Наша цель – уменьшить число SQL-выражений и упростить их, чтобы запросы выполнялись как можно быстрее.

Рассмотрим план извлечения, рекомендованный нами выше в этой главе: каждая связь и коллекция должна загружаться по требованию. Такой план почти наверняка повлечет за собой слишком большое количество SQL-запросов, каждый из которых будет загружать лишь небольшой объем данных. Этот план ведет к *проблеме n + 1 выражений SELECT*, поэтому сначала обсудим эту проблему. Альтернативный план извлечения, использующий немедленную загрузку, потребует меньше SQL-запросов, поскольку каждый запрос будет загружать в память больше данных. С увеличением объемов данных, загружаемых запросами, вы можете столкнуться с *проблемой декартова произведения*.

Вам придется найти золотую середину между этими двумя крайностями – идеальную стратегию извлечения для каждой процедуры и каждого сценария использования в вашем приложении. Так же как для планов извлечения, глобальную стратегию извлечения можно задать для отображений: определить настройки по умолчанию, действующие всегда. Затем можете переопределить стратегию извлечения по умолчанию для конкретной процедуры, используя произвольные запросы JPQL, `CriteriaQuery` или даже SQL.

Для начала обсудим фундаментальные проблемы, с которыми можно столкнуться, начав с проблемы *n + 1 выражений SELECT*.

12.2.1. Проблема n + 1 выражений SELECT

Эту проблему проще продемонстрировать на примере. Предположим, что вы используете план отложенного извлечения в отображении, и все данные загружают-

ся только по мере необходимости. Следующий пример проверяет, задано ли имя пользователя (username) у каждого продавца (seller) товара (Item):

Файл: /examples/src/test/java/org/jpwh/test/fetching/NPlusOneSelects.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
    // select * from USERS where ID = ?
}
```

Каждый продавец будет загружен дополнительной инструкцией SELECT

Как видите, одно SQL-выражение SELECT загружает экземпляры сущностей Item. Затем, во время обхода всех товаров (items), для получения каждого экземпляра User требуется выполнить дополнительные инструкции SELECT. Это тождественно одному запросу для экземпляра Item и *n* дополнительным запросам в зависимости от количества товаров, а также от того, продает ли конкретный пользователь (User) более одного товара (Item). Это явно не самая эффективная стратегия, если заранее известно, что придется обращаться к полю seller каждого экземпляра Item.

С той же проблемой можно столкнуться, применив отложенную загрузку для коллекций. Следующий пример проверяет количество ставок (bids) для каждого товара (Item):

Файл: /examples/src/test/java/org/jpwh/test/fetching/NPlusOneSelects.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID = ?
}
```

Для загрузки каждой коллекции bids требуется выполнить дополнительный запрос SELECT

Если заранее известно, что придется обращаться ко всем коллекциям bids, загрузка каждой из них по отдельности снова оказывается неэффективной. Если у вас имеется 100 товаров, выполнится 101 запрос SQL!

С теми знаниями, которыми вы уже владеете, вам, возможно, захочется поменять план извлечения по умолчанию в своих отображениях и указать параметр FetchType.EAGER для связей seller или bids. Но, поступив так, вы столкнетесь с проблемой *декартова произведения*.

12.2.2. Проблема декартова произведения

Исследовав предметную модель и модель данных и обнаружив, что каждый раз при загрузке экземпляра Item также требуется загружать его свойство seller, вы можете решить отобразить связь с параметром FetchType.EAGER. Если вам нужна гарантия, что при этом с каждым экземпляром Item будет загружаться его свойство seller, нужно обеспечить доступность этих данных в отсоединенном состоянии экземпляра Item и при закрытом контексте хранения:

Файл: /model/src/main/java/org/jpwh/model/fetching/cartesianproduct/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)
    protected User seller;

    // ...
}
```

Для реализации немедленного извлечения Hibernate использует SQL-предложение `JOIN`, чтобы загрузить экземпляры `Item` и `User` в одном запросе `SELECT`:

```
item = em.find(Item.class, ITEM_ID);
// select i.*, u.*
//  from ITEM i
//   left outer join USERS u on u.ID = i.SELLER_ID
//  where i.ID = ?
```

Результат запроса будет содержать одну запись с данными из таблицы `ITEM`, объединенную с данными из таблицы `USERS`, как показано на рис. 12.4.

i.ID	i.NAME	i.SELLER_ID	...	u.ID	u.USERNAME	...
1	One	2	...	2	johndoe	...

Рис. 12.4 ❖ Hibernate соединяет две таблицы, чтобы обеспечить немедленную загрузку связанных записей

Немедленное извлечение, по умолчанию использующее стратегию `JOIN`, не вызовет проблем для связей `@ManyToOne` и `@OneToOne`. В одном запросе SQL с несколькими предложениями `JOIN` можно одновременно загрузить товар (`Item`), его продавца (`seller`), адрес (`Address`) пользователя (`User`), его город (`City`) и т. д. Даже при отображении этих связей с параметром `FetchType.EAGER` запрос вернет только одну запись. Но в *какой-то* момент Hibernate должен перестать следовать плану `FetchType.EAGER`. Количество соединяемых таблиц зависит от глобального параметра конфигурации `hibernate.max_fetch_depth`. По умолчанию никаких ограничений не задано. Приемлемыми являются значения от 1 до 5. Вы даже можете отключить извлечение связей `@ManyToOne` и `@OneToOne` с использованием `JOIN`, установив значение 0. При достижении лимита Hibernate будет попрежнему немедленно загружать данные в соответствии с планом извлечения, но используя дополнительные выражения `SELECT` (обратите внимание, что этот параметр можно задать в некоторых диалектах баз данных: например, в диалекте `MySQL` он равен 2).

Немедленная загрузка коллекций с помощью `JOIN`, с другой стороны, может привести к серьезному падению производительности. Если стратегию `FetchType.EAGER` применить также к коллекциям `bids` и `images`, возникнет *проблема декартова произведения*.

Эта проблема появляется, когда одним запросом SQL с предложением JOIN одновременно загружаются две коллекции. Давайте создадим такой план извлечения, а затем исследуем проблему:

Файл: /model/src/main/java/org/jpwh/model/fetching/cartesianproduct/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
    protected Set<Bid> bids = new HashSet<>();

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    protected Set<String> images = new HashSet<String>();

    // ...
}
```

Не имеет значения, отмечены ли обе коллекции аннотациями @OneToMany, @ManyToMany или @ElementCollection. Независимо от содержимого коллекции, немедленная загрузка более чем одной коллекции с применением SQL-предложения JOIN является фундаментальной проблемой. При загрузке экземпляра Item Hibernate выполнит проблемное выражение SQL:

Файл: /examples/src/test/java/org/jpwh/test/fetching/CartesianProduct.java

```
Item item = em.find(Item.class, ITEM_ID);
// select i.*, b.*, img.*
//  from ITEM i
//   left outer join BID b on b.ITEM_ID = i.ID
//   left outer join IMAGE img on img.ITEM_ID = i.ID
// where i.ID = ?

em.detach(item);

assertEquals(item.getImages().size(), 3);
assertEquals(item.getBids().size(), 3);
```

Как видите, Hibernate последовал нашему плану извлечения, и теперь есть возможность обращаться к коллекциям bids и images в отсоединенном состоянии. Проблема лишь в том, *как* эти коллекции загружаются – с помощью SQL-операции JOIN, результатом которой является прямое произведение. Рассмотрим результат запроса на рис. 12.5.

Он содержит много лишних данных, тогда как фреймворку необходимы лишь выделенные ячейки. Экземпляр Item имеет три ставки и три изображения. Мощность произведения зависит от размеров извлекаемых коллекций: трижды три равняется девяти итоговым записям. Представьте теперь, что имеется экземпляр Item с 50 ставкам (bids) и 5 изображениями (images); в этом случае запрос вернет 250 записей! При создании собственных запросов JPQL или CriteriaQuery можно

получить еще более впечатляющие результаты: представьте, что произойдет при извлечении 500 товаров и немедленной загрузке с помощью `JOIN` десятков ставок и изображений.

i.ID	i.NAME	...	b.ID	b.AMOUNT	img.FILENAME
1	One	...	1	99.00	foo.jpg
1	One	...	1	99.00	bar.jpg
1	One	...	1	99.00	baz.jpg
1	One	...	2	100.00	foo.jpg
1	One	...	2	100.00	bar.jpg
1	One	...	2	100.00	baz.jp
1	One	...	3	101.00	foo.jpg
1	One	...	3	101.00	bar.jpg
1	One	...	3	101.00	baz.jpg

Рис. 12.5 ❖ Результатом двух соединений нескольких строк является прямое произведение

Для создания таких результатов серверу базы данных потребуется значительный объем памяти и процессорного времени. Если вы надеетесь, что драйвер JDBC каким-то образом выполнит сжатие данных, вы, вероятно, ожидаете от создателей базы данных слишком многого. Преобразуя результаты запроса в хранимые экземпляры и коллекции, Hibernate сразу избавится от всех дубликатов; информация в невыделенных ячейках на рис. 12.5 будет проигнорирована. Очевидно, вы не сможете удалить эти дубликаты на уровне SQL – оператор `DISTINCT` здесь не поможет.

Можно было бы одновременно извлечь экземпляр сущности и две коллекции быстрее, если вместо одного запроса SQL с огромным результатом выполнить три отдельных запроса. Далее мы рассмотрим этот тип оптимизации, а также способы поиска и реализации лучшей стратегии извлечения. Мы заново рассмотрим план отложенного извлечения по умолчанию и попробуем решить проблему *n + 1 выражений SELECT*.

12.2.3. Массовая предварительная выборка данных

Если Hibernate будет извлекать все связи сущностей и коллекции по требованию, для выполнения конкретной процедуры может понадобиться множество дополнительных SQL-выражений `SELECT`. Как и прежде, рассмотрим код, проверяющий наличие имени пользователя (`username`) у поставщика (`seller`) каждого товара (`Item`). При использовании отложенной загрузки это потребует одной инструкции `SELECT`, чтобы получить все экземпляры `Item`, и еще *n* инструкций `SELECT` для инициализации прокси-объекта `seller` каждого экземпляра `Item`.

Hibernate реализует несколько алгоритмов, способных производить предварительную выборку данных. Первый алгоритм, который мы обсудим, – *массовое извлечение* (batch fetching), он работает следующим образом: если Hibernate должен инициализировать один прокси-объект `User`, тогда в этой же инструкции `SELECT` он инициализирует еще несколько. Другими словами, если известно, что в контексте хранения находится несколько экземпляров `Item` и в каждом связь `seller` ссылается на прокси-объект, тогда при обращении к базе данных можно инициализировать сразу несколько прокси-объектов.

Посмотрим, как это работает. Во-первых, нужно включить массовое извлечение экземпляров `User` с помощью аннотации:

Файл: /model/src/main/java/org/jpwh/model/fetching/batch/User.java

```
@Entity
@org.hibernate.annotations.BatchSize(size = 10)
@Table(name = "USERS")
public class User {
    // ...
}
```

Эта настройка сообщает Hibernate, что при загрузке одного прокси-объекта `User` он может в одной инструкции `SELECT` загрузить до 10 прокси-объектов. Массовое извлечение обычно называют *оптимизацией вслепую*, поскольку неизвестно, как много неинициализированных прокси-объектов `User` может находиться в конкретном контексте хранения. Нельзя с уверенностью утверждать, что 10 – это идеальное значение, – это лишь предположение. Зато вы знаете, что вместо $n + 1$ запросов SQL будет выполнено $n + 1/10$ запросов, что является существенным улучшением. Приемлемые значения обычно невелики, поскольку редко бывает желательно загружать в память слишком много данных, особенно если нет уверенности, что они понадобятся.

Следующая оптимизированная процедура проверяет имя пользователя (`username`) каждого продавца (`seller`):

Файл: /examples/src/test/java/org/jpwh/test/fetching/Batch.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
    // select * from USERS where ID in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
}
```

Обратите внимание на запрос SQL, который будет выполнен перед началом обхода коллекции `items`. В первом вызове `item.getSeller().getUserName()` Hibernate должен инициализировать первый прокси-объект `User`. Вместо загрузки одной записи из таблицы `USERS` Hibernate извлечет несколько записей и загрузит до 10 экземпляров `User`. Как только вы обратитесь к одиннадцатому полю `seller`, тут

же будет загружено еще 10 экземпляров; так будет продолжаться до тех пор, пока в контексте хранения не останется неинициализированных прокси-объектов `User`.

ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ:
Как на самом деле работает алгоритм массового извлечения?

Мы несколько упростили описание алгоритма массовой загрузки. На практике он действует немного иначе. В качестве примера рассмотрим ситуацию, когда пакет имеет размер 32. При запуске Hibernate неявно создаст несколько массовых загрузчиков. Каждый загрузчик знает, сколько прокси-объектов он может инициализировать: 32, 16, 10, 9, 8, 7, ..., 1. Цель состоит в том, чтобы минимизировать потребление памяти для создания загрузчиков и создать достаточное их количество, чтобы можно было выполнить массовую загрузку с любым размером пакета. Очевидно, другая цель состоит в минимизации количества запросов SQL.

Для инициализации 31 прокси-объекта Hibernate загрузит 3 набора (возможно, вы ожидали 1, поскольку 32 > 31). Hibernate автоматически выберет загрузчики с размерами наборов 16, 10 и 5. Вы можете настроить работу алгоритма массового извлечения с помощью параметра `hibernate.batch_fetch_style` в конфигурации единицы хранения. Значением по умолчанию является `LEGACY`, что приводит к созданию и выбору нескольких загрузчиков во время запуска. Другими вариантами являются `PADDED` и `DYNAMIC`. При использовании параметра `PADDED` Hibernate создаст только один запрос SQL для массового загрузчика с 32 свободными аргументами в предложении `IN`, и если нужно будет загрузить меньше 32 прокси-объектов, некоторые значения идентификаторов будут повторяться. При использовании параметра `DYNAMIC` Hibernate динамически сформирует SQL-выражение для массовой загрузки во время выполнения, когда ему будет известно количество прокси-объектов для инициализации.

Массовое извлечение также доступно для коллекций:

Файл: /model/src/main/java/org/jpwh/model/fetching/batch/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.BatchSize(size = 5)
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Теперь при инициализации одной коллекции `bids` будет загружено до пяти дополнительных коллекций `Item#bids`, если они еще не инициализированы в текущем контексте хранения.

Файл: /examples/src/test/java/org/jpwh/test/fetching/Batch.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
```

```
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID in (?, ?, ?, ?, ?)
}
```

При первом вызове `item.getBids().size()` во время обхода будет предварительно загружен целый набор коллекций объектов `Bid` для других экземпляров `Item`.

Массовое извлечение – это простая и, как правило, разумная оптимизация, способная значительно сократить количество выражений SQL, которые в противном случае понадобилось бы выполнить для инициализации прокси-объектов и коллекций. При этом есть риск загрузить данные, которые затем не понадобятся, и потратить больше памяти, однако уменьшение количества обращений к базе данных может оказать существенное влияние. Память обходится дешево, а масштабирование серверов баз данных – нет.

Другой алгоритм предварительного извлечения, который не является оптимизацией вслепую, использует подзапросы для инициализации нескольких коллекций.

Особенности Hibernate

12.2.4. Предварительное извлечение коллекций с помощью подзапросов

Возможно, более эффективной стратегией загрузки всех коллекций `bids` для нескольких экземпляров `Item` является предварительное извлечение с помощью подзапросов. Для применения этой оптимизации отметьте коллекцию аннотацией Hibernate:

Файл: /model/src/main/java/org/jpwh/model/fetching/subselect/Item.java

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SUBSELECT
    )
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Теперь, как только вы инициализируете одну из коллекций `bids`, Hibernate инициализирует все коллекции `bids` для всех загруженных экземпляров `Item`.

Файл: /examples/src/test/java/org/jpwh/test/fetching/Subselect.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID in (
```

```
    // select ID from ITEM
    // )
}
```

Hibernate запомнит первоначальный запрос, использовавшийся для загрузки коллекции `items`. Затем встроит этот запрос (немного измененный) внутрь подзапроса, чтобы извлечь коллекцию `bids` для каждого экземпляра `Item`.

Предварительное извлечение с применением подзапросов является мощной оптимизацией, но на момент написания этих строк она была доступна только для коллекций с планом отложенной загрузки, но не для прокси-объектов сущностей. Также обратите внимание, что Hibernate запомнит первоначальный запрос, используемый затем в подзапросе, только в рамках конкретного контекста хранения. Если отсоединить экземпляр `Item`, не инициализировав коллекцию `bids`, а затем выполнить слияние с новым контекстом хранения и начать обход коллекции, предварительного извлечения остальных коллекций не произойдет.

При следовании глобальному плану отложенного извлечения предварительное извлечение с помощью подзапросов и массовое извлечение сокращают количество запросов, помогая справиться с *проблемой n + 1 запросов SELECT*. Если вместо этого для связей и коллекций использовать глобальный план отложенного извлечения, вам придется решать *проблему декартова произведения*, разбивая запрос с несколькими предложениями `JOIN` на несколько выражений `SELECT`.

Особенности Hibernate

12.2.5. Отложенное извлечение с несколькими выражениями SELECT

При попытке извлечь несколько коллекций в одном запросе SQL с несколькими предложениями `JOIN` вы столкнетесь с *проблемой декартова произведения*, как было показано выше. Однако есть возможность попросить Hibernate не использовать операцию `JOIN`, а загружать данные с помощью дополнительных инструкций `SELECT`, чтобы избежать слишком объемных результатов запросов и прямых произведений SQL с дубликатами:

Файл: /model/src/main/java/org/jpwh/model/fetching/eagerselect/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT
    )
    protected User seller;

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT
```

По умолчанию JOIN

```
    )
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Теперь вместе с экземпляром `Item` будут загружены поля `seller` и `bids`:

Файл: /examples/src/test/java/org/jpwh/test/fetching/EagerSelect.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?
// select * from USERS where ID = ?
// select * from BID where ITEM_ID = ?

em.detach(item);

assertEquals(item.getBids().size(), 3);
assertNotNull(item.getBids().iterator().next().getAmount());
assertEquals(item.getSeller().getUsername(), "johndoe");
```

Чтобы загрузить запись из таблицы `ITEM`, Hibernate выполнит одну инструкцию `SELECT`. Затем он сразу же выполнит еще две инструкции `SELECT`: одна загрузит запись из таблицы `USERS` (поле `seller`), а вторая – несколько записей из таблицы `BID` (коллекция `bids`).

Дополнительные выражения `SELECT` выполняются сразу же; метод `find()` выполнит несколько запросов SQL. Видно, что Hibernate следует плану немедленного извлечения – все данные будут доступны в отсоединенном состоянии.

Все эти настройки действуют глобально – они работают всегда. Опасность заключается в том, что изменение одной настройки для конкретной проблемной ситуации в вашем приложении может оказать негативное влияние на другие процедуры. Поддерживать баланс непросто, поэтому мы советуем отображать все связи сущности с параметром `FetchType.LAZY`, как было показано ранее.

Гораздо практичнее применять немедленное извлечение и операции `JOIN` *динамически*, отдельно для конкретной процедуры.

12.2.6. Динамическое немедленное извлечение

Так же, как в предыдущих разделах, представим, что нужно проверить свойство `username` в каждом объекте `Item#seller`. Используя глобальный план отложенного извлечения, можно загрузить данные, необходимые для этой процедуры, а затем *динамически* применить стратегию немедленного извлечения:

Файл: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
List<Item> items =
    em.createQuery("select i from Item i join fetch i.seller")
        .getResultList();
```

```
// select i.*, u.*
//  from ITEM i
//   inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?

em.close();  ← Отсоединить все

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
}
```

Самые важные ключевые слова в этом запросе JPQL – `join fetch` – требуют от Hibernate применить SQL-операцию `JOIN` (фактически `INNER JOIN`) для получения свойства `seller` каждого экземпляра `Item` в том же запросе. Тот же самый запрос можно выразить с помощью `CriteriaQuery` API:

Файл: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();

Root<Item> i = criteria.from(Item.class);
i.fetch("seller");
criteria.select(i);

List<Item> items = em.createQuery(criteria).getResultList();

em.close();  ← Отсоединить все

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
}
```

Динамическое немедленное извлечение с соединениями также можно применять к коллекциям. Следующий код загружает все объекты из коллекций `bids` всех экземпляров `Item`:

Файл: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
List<Item> items =
    em.createQuery("select i from Item i left join fetch i.bids")
        .getResultList();
// select i.*, b.*
//  from ITEM i
//   left outer join BID b on b.ITEM_ID = i.ID
// where i.ID = ?

em.close();  ← Отсоединить все

for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
}
```

Теперь то же самое, но с `CriteriaQuery` API:

Файл: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();

Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
criteria.select(i);

List<Item> items = em.createQuery(criteria).getResultList();

em.close();  ← Отсоединить все

for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
}
```

Обратите внимание, что для извлечения коллекций требуется выражение `LEFT OUTER JOIN`, поскольку вам также понадобятся записи из таблицы `ITEM`, даже если не найдется соответствующей коллекции `bids`. Мы еще многое расскажем про извлечение данных с помощью JPQL и `CriteriaQuery` далее, в главе 15. Там вы увидите много примеров внутренних, внешних, левых и правых соединений, так что не беспокойтесь пока об этих особенностях.

Для динамического переопределения глобального плана извлечения в предметной модели необязательно писать запросы вручную. Также можно использовать декларативный подход, определяя *профили извлечения*.

12.3. Профили извлечения

Профили извлечения дополняют параметры извлечения в языке запросов и различных API. Они позволяют управлять определениями профилей с помощью XML и метаданных в аннотациях. Ранние версии Hibernate не поддерживали специальных профилей извлечения, но сегодня Hibernate поддерживает следующие возможности.

- *Профили извлечения* – нестандартный API, основанный на декларативном определении профилей с помощью аннотации `@org.hibernate.annotations.FetchProfile` и метода `Session#enableFetchProfile()`. На текущий момент этот простой механизм поддерживает выборочное переопределение отображаемых ассоциаций сущностей с отложенной загрузкой, активируя стратегию немедленного извлечения с применением `JOIN` в конкретной единице работы.
- *Графы сущностей* – как определено в JPA 2.1, граф сущностей и связей можно определить с помощью аннотации `@EntityGraph`. Этот план извлечения (или комбинация планов) может активироваться с помощью подсказки при вызове `EntityManager#find()` или выполнении запросов (JPQL, запросов с критериями). Данный граф определяет, *что* должно быть загружено; к сожалению, он не определяет, *как*.

Справедливости ради следует сказать, что здесь есть что улучшить, и мы ожидаем от будущих версий Hibernate и JPA более единообразного и мощного API.

Не забывайте, что выражения JPQL и SQL можно выносить во внешние файлы метаданных (см. раздел 14.4). Запрос JPQL *является* декларативным (именованным) профилем извлечения; чего ему не хватает, так это возможности легко накладывать различные планы на один базовый запрос. Мы сталкивались с довольно интересными решениями, основанными на манипуляции строками, но их лучше избегать. С другой стороны, используя запросы с критериями, вы получаете всю доступную мощь Java для организации кода построения запросов. Кроме того, графы сущностей позволяют повторно применять план извлечения к любому типу запроса.

Давайте сначала обсудим профили извлечения Hibernate и способы переопределения глобального плана отложенного извлечения для конкретной единицы работы.

Особенности Hibernate

12.3.1. Определение профилей извлечения Hibernate

Профили извлечения Hibernate – это глобальные метаданные, которые определяются для целой единицы хранения. Можно, конечно, поместить аннотацию `@FetchProfile` перед классом, но мы предпочитаем метаданные уровня пакета в файле `package-info.java`:

Файл: /model/src/main/java/org/jpwh/model/fetching/profile/package-info.java

```
@org.hibernate.annotations.FetchProfiles({
    @FetchProfile(name = Item.PROFILE_JOIN_SELLER,      ← ❶ Название профиля
        fetchOverrides = @FetchProfile.FetchOverride(   ← ❷ Переопределение
            entity = Item.class,
            association = "seller",
            mode = FetchMode.JOIN  ← ❸ Режим JOIN
        )),

    @FetchProfile(name = Item.PROFILE_JOIN_BIDS,
        fetchOverrides = @FetchProfile.FetchOverride(
            entity = Item.class,
            association = "bids",
            mode = FetchMode.JOIN
        ))
})
```

❶ Каждый профиль имеет имя. Это просто строка, представленная константой.
❷ Каждое переопределение в профиле указывает на одну связь или коллекцию.
❸ На момент написания этих строк был доступен только режим JOIN.

Теперь профили можно подключать к конкретной единице работы:

Файл: /examples/src/test/java/org/jpwh/test/fetching/Profile.java

```
Item item = em.find(Item.class, ITEM_ID);  ← ❶ Извлечение экземпляра

em.clear();
```

```
em.unwrap(Session.class).enableFetchProfile(Item.PROFILE_JOIN_SELLER);
item = em.find(Item.class, ITEM_ID);

em.clear();
em.unwrap(Session.class).enableFetchProfile(Item.PROFILE_JOIN_BIDS);
item = em.find(Item.class, ITEM_ID);
```

❷ Подключает профиль

❸ Накладывается на второй профиль

❶ Для поля `Item#seller` задано отложенное отображение, поэтому с планом извлечения по умолчанию будет получен только экземпляр сущности `Item`.

❷ Для подключения профиля нужен Hibernate API. После этого он будет использоваться для любой операции в рамках этой единицы работы. Поле `Item#seller` загружается при помощи соединения в том же запросе SQL, всякий раз, когда экземпляр `Item` загружается этим экземпляром `EntityManager`.

❸ На ту же единицу работы можно наложить другой профиль. Теперь при загрузке каждого экземпляра `Item` поле `Item#seller` и коллекция `Item#bids` будут загружены в одном запросе SQL с соединением.

Несмотря на простоту, профили извлечения Hibernate могут оказаться действенным решением проблем оптимизации извлечения в небольших или простых приложениях. С появлением JPA 2.1 и *графов сущностей* похожая функциональность появилась в стандартизованном виде.

12.3.2. Графы сущностей

Граф сущностей – это объявление экземпляров и атрибутов, переопределяющее или улучшающее план извлечения по умолчанию при вызове `EntityManager#find()` или с помощью подсказки в операциях с запросами. Следующий пример демонстрирует извлечение с использованием графа сущностей:

Файл: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
Map<String, Object> properties = new HashMap<>();
properties.put(
    "javax.persistence.loadgraph",
    em.getEntityGraph(Item.class.getSimpleName())
);

Item item = em.find(Item.class, ITEM_ID, properties);
// select * from ITEM where ID = ?
```

Строка «Item»

Используемый граф сущностей имеет имя `Item`, а подсказка для операции `find()` указывает, что это еще и *граф загрузки*. Это означает, что все атрибуты, указанные как вершины графа сущностей, будут загружаться с параметром `FetchType.EAGER`, а те, что не указаны, будут загружаться соответственно их параметрам или типу `FetchType`, заданному в отображении по умолчанию.

Ниже представлено определение этого графа, а также плана извлечения класса сущности по умолчанию:

Файл: /model/src/main/java/org/jpwh/model/fetching/fetchloadgraph/Item.java

```
@NamedEntityGraphs({
    @NamedEntityGraph  ← По умолчанию используется граф сущностей «Item»
})
@Entity
public class Item {

    @NotNull
    @ManyToOne(fetch = FetchType.LAZY)
    protected User seller;

    @OneToMany(mappedBy = "item")
    protected Set<Bid> bids = new HashSet<>();

    @ElementCollection
    protected Set<String> images = new HashSet<>();

    // ...
}
```

В метаданных графы сущностей имеют названия и связаны с классами сущностей; обычно они объявляются с помощью аннотаций перед классами сущностей. При желании объявление можно поместить в файл XML. Если явно не указать имя графа, он получит простое имя класса сущности-владельца: в данном случае это `Item`. Если не перечислить вершины графа с указанием атрибутов, как в случае с пустым графом сущностей из предыдущего примера, по умолчанию будут использоваться параметры класса сущности. В классе `Item` все связи и коллекции отображаются как отложенные; это план извлечения по умолчанию. То есть все, что мы сделали, не даст никакого результата, и операция `find()` безо всяких подсказок даст такой же результат: загрузит только экземпляр `Item`, но не поля `seller`, `bids` и `images`.

Другим вариантом является построение графа сущностей с помощью API:

Файл: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
EntityGraph<Item> itemGraph = em.createEntityGraph(Item.class);

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", itemGraph);

Item item = em.find(Item.class, ITEM_ID, properties);
```

Это снова пустой граф сущностей без вершин с атрибутами, передаваемый прямо в операцию извлечения.

Предположим, что требуется создать граф сущностей, который меняет отложенное извлечение по умолчанию для поля `Item#seller` на немедленное:

Файл: /model/src/main/java/org\jpwh/model/fetching/fetchloadgraph/Item.java

```
@NamedEntityGraphs({
    @NamedEntityGraph(
        name = "ItemSeller",
        attributeNodes = {
```

```
            @NamedAttributeNode("seller")
        }
    )
})
@Entity
public class Item {

    // ...
}
```

Теперь этот граф можно использовать, когда потребуется немедленная загрузка экземпляра `Item` и поля `seller`:

Файл: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
Map<String, Object> properties = new HashMap<>();
properties.put(
    "javax.persistence.loadgraph",
    em.getEntityGraph("ItemSeller")
);

Item item = em.find(Item.class, ITEM_ID, properties);
// select i.*, u.*
//  from ITEM i
//   inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

Если нежелательно жестко определять граф с помощью аннотаций, его можно создать, используя API:

```
EntityGraph<Item> itemGraph = em.createEntityGraph(Item.class);
itemGraph.addAttributeNodes(Item_.seller);  ← Статическая метамодель

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", itemGraph);

Item item = em.find(Item.class, ITEM_ID, properties);
// select i.*, u.*
//  from ITEM i
//   inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

До сих пор мы применяли подсказки только в операции `find()`. Графы сущностей тоже могут использоваться как подкаски в запросах:

Файл: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
List<Item> items =
    em.createQuery("select i from Item i")
        .setHint("javax.persistence.loadgraph", itemGraph)
        .getResultList();
// select i.*, u.*
//  from ITEM i
//   left outer join USERS u on u.ID = i.SELLER_ID
```

Графы сущностей могут быть сложными. Следующий пример демонстрирует, как работать с повторно используемыми объявлениями подграфов:

Файл: /model/src/main/java/org/jpwh/model/fetching/fetchloadgraph/Bid.java

```
@NamedEntityGraphs({
    @NamedEntityGraph(
        name = "BidBidderItemSellerBids",
        attributeNodes = {
            @NamedAttributeNode(value = "bidder"),
            @NamedAttributeNode(
                value = "item",
                subgraph = "ItemSellerBids"
            )
        },
        subgraphs = {
            @NamedSubgraph(
                name = "ItemSellerBids",
                attributeNodes = {
                    @NamedAttributeNode("seller"),
                    @NamedAttributeNode("bids")
                })
        }
    )
})
@Entity
public class Bid {
    // ...
}
```

При использовании этого графа сущностей в качестве графа загрузки получение экземпляров `Bid` повлечет за собой извлечение полей `Bid#bidder`, `Bid#item`, а также `Item#seller` и всей коллекции `Item#bids`. Несмотря на то что графам сущностей можно давать любые имена, мы советуем выработать соглашение, которому сможет следовать каждый член вашей команды, а также вынести строки в общие константы.

При работе с API графов сущностей предыдущий план извлечения выглядит так:

Файл: /examples/src/test/java/org/jpwh/test/fetching/FetchLoadGraph.java

```
EntityGraph<Bid> bidGraph = em.createEntityGraph(Bid.class);
bidGraph.addAttributeNodes(Bid_.bidder, Bid_.item);
Subgraph<Item> itemGraph = bidGraph.addSubgraph(Bid_.item);
itemGraph.addAttributeNodes(Item_.seller, Item_.bids);

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", bidGraph);

Bid bid = em.find(Bid.class, BID_ID, properties);
```

До сих пор мы использовали графы сущностей в качестве графов *загрузки*. Но граф сущностей также можно использовать в качестве *графа извлечения* с помощью подсказки `javax.persistence.fetchgraph`. При вызове метода `find()` или выполнении запроса с использованием графа извлечения любые атрибуты и коллекции, не указанные в плане, будут загружены с параметром `FetchType.LAZY`, а указанные – с параметром `FetchType.EAGER`. Фактически это отменяет действие всех параметров `FetchType` для атрибутов сущности и отображений коллекций, в то время как граф загрузки вносит только дополнения.

Следует отметить две слабые стороны графов сущностей JPA, с которыми вы столкнетесь довольно быстро. Во-первых, нельзя менять только планы извлечения, без изменения стратегии извлечения (массовая загрузка /подзапросы/объединения/выборки). Во-вторых, определение графа сущностей с помощью аннотаций или XML небезопасно с точки зрения типов: имена атрибутов задаются строками. Интерфейс `EntityGraph`, напротив, является типобезопасным.

12.4. Резюме

- Профиль извлечения объединяет план извлечения (какие данные загружать) со стратегией извлечения (как их загружать) и определяется в метаданных или аннотациях, которые можно использовать повторно.
- Мы создали глобальный план извлечения и определили, какие связи и коллекции должны всегда загружаться в память. Мы создали план извлечения, основанный на вариантах использования, определяющий способы доступа к связанным сущностям и обхода коллекций, а также наборы данных, которые должны быть доступны в отсоединенном состоянии.
- Научились выбирать правильную стратегию для плана извлечения. Вашей целью должны быть минимизация количества выражений SQL и снижение сложности отдельных выполняемых выражений SQL. Особенно следует избегать проблем *n + 1 выражений SELECT* и декартова произведения, которые мы подробно обсудили, используя различные стратегии оптимизации.
- Вы познакомились с профилями извлечения Hibernate и графами сущностей, а также с профилями извлечения JPA.

Глава 13

Фильтрация данных

В этой главе:
- каскадная передача изменений состояния;
- перехват и обработка событий;
- аудит и версионирование с помощью Hibernate Envers;
- динамическая фильтрация данных.

В этой главе вы познакомитесь со множеством стратегий *фильтрации* данных при их прохождении сквозь механизмы Hibernate. Когда Hibernate загружает информацию из базы данных, при помощи фильтра можно прозрачно отсекать данные, которые не должны быть доступны приложению. Также можно перехватывать такие события, как запись информации в базу данных, и запускать вспомогательные процедуры: например, делать записи в журнале аудита или присваивать записям виртуальные идентификаторы.

Мы рассмотрим следующие возможности фильтрации данных и API.

- В разделе 13.1 вы узнаете, как реагировать на изменения состояния экземпляра сущности и *каскадно изменять состояния* связанных сущностей. Например, при сохранении объекта `User` (пользователь) Hibernate может автоматически сохранить все связанные с ним объекты `BillingDetails` (платежные реквизиты). При удалении товара (`Item`) Hibernate может удалить все связанные с ним экземпляры ставок (`Bid`). Вы можете воспользоваться этой стандартной функциональностью JPA с помощью специальных атрибутов отображения связей и коллекций.
- Стандарт Java Persistence определяет поддержку *методов обратного вызовы* и *приемников событий* жизненного цикла. Приемник событий (event listener) – это класс со специальными методами, которые Hibernate вызывает при изменении состояния экземпляра сущности: например, после загрузки из базы данных или перед удалением из нее. Эти методы обратного вызова могут также определяться в классах сущностей, с помощью специальных аннотаций. Это дает возможность производить дополнительные побочные эффекты при изменении состояния. В Hibernate имеется также несколько нестандартных механизмов, позволяющих перехватывать события жизненного цикла на более низком уровне, которые мы обсудим в разделе 13.2.

- Типичным побочным эффектом является запись в *журнал аудита*; подобный журнал обычно содержит информацию о том, какие данные изменились, когда были сделаны изменения и кто их сделал. Более продвинутые системы аудита могут хранить несколько версий данных, а также *временные представления* (temporal views); при желании можно запросить у Hibernate данные в том состоянии, в каком они были, например, неделю назад. Это сложная проблема, и в разделе 13.3 мы познакомим вас с *Hibernate Envers* – подпроектом, задачей которого являются версионирование и аудит в приложениях JPA.
- В разделе 13.4 вы узнаете, что в Hibernate API имеются также *фильтры данных* (data filters), позволяющие добавлять произвольные ограничения в SQL-выражения `SELECT`, которые выполняет Hibernate. Благодаря им можно определить произвольное ограниченное представление данных на уровне приложения. К примеру, можно применить фильтр, ограничивающий данные по региону продажи или по критерию авторизации.

Сначала рассмотрим варианты каскадирования изменений состояния.

Главные нововведения в JPA 2

- Добавлена поддержка внедрения зависимостей с помощью CDI в классах приемников событий сущностей JPA.

13.1. Каскадная передача изменений состояния

Когда состояние экземпляра сущности изменяется – например, при переходе из *временного* состояния в *хранимое*, – связанные экземпляры сущностей также могут стать частью этого изменения. Подобная *каскадная* передача изменений по умолчанию выключена; каждый экземпляр сущности имеет свой, независимый жизненный цикл. Но для некоторых связей между сущностями может понадобиться тонкая настройки зависимостей жизненных циклов.

Например, в разделе 7.3 мы определили связь между классами сущностей `Item` (товар) и `Bid` (ставка). В этом случае ставки за товар (`Item`) автоматически становились хранимыми при добавлении в коллекцию экземпляра `Item`, а при удалении сущности-владельца `Item` они автоматически удалялись. Фактически мы сделали класс сущности `Bid` зависимым от другой сущности `Item`.

В отображении этой связи мы использовали настройки каскадирования `CascadeType.PERSIST` и `CascadeType.REMOVE`. Мы также обсудили специальный параметр `orphanRemoval` и то, как каскадное удаление на уровне базы данных (с применением параметра внешнего ключа `ON DELETE`) влияет на приложение.

Вам стоит заново просмотреть эти отображения связей, поскольку мы не станем повторять их здесь. В этом разделе мы рассмотрим другие, более редкие способы каскадирования.

13.1.1. Доступные способы каскадирования

Все способы каскадирования, доступные в Hibernate, перечислены в табл. 13.1. Обратите внимание, как каждый из них связан с операциями `EntityManager` или `Session`.

Таблица 13.1. Способы каскадирования для отображения связей между сущностями

Параметр	Описание
`CascadeType.PERSIST`	При сохранении экземпляра сущности с помощью метода `EntityManager #persist()` любой связанный экземпляр сущности также перейдет в хранимое состояние во время выталкивания контекста
`CascadeType.REMOVE`	При удалении экземпляра сущности с помощью метода `EntityManager #remove()` любой связанный экземпляр сущности также будет удален удален во время выталкивания контекста
`CascadeType.DETACH`	При отсоединении экземпляра сущности от контекста хранения с помощью `EntityManager#detach()` любой ассоциированный экземпляр сущности также будет отсоединен
`CascadeType.MERGE`	При слиянии временной или отсоединенной сущности с контекстом персистентности с помощью `EntityManager#merge()` для любого связанного временного или отсоединенного экземпляра сущности также будет выполнено слияние
`CascadeType.REFRESH`	При изменении экземпляра сущности с помощью `EntityManager#refresh()` любой связанный экземпляр сущности также будет изменен
`org.hibernate.annotations .CascadeType.REPLICATE`	При копировании отсоединенного экземпляра сущности в базу данных с помощью `Session#replicate()` любой связанный отсоединенный экземпляр сущности также будет скопирован
`CascadeType.ALL`	Сокращенная запись для применения всех способов каскадирования к отображаемой связывания

Если вам интересно, дополнительные способы каскадирования можно найти в перечислении `org.hibernate.annotations.CascadeType`. Но на данный момент нас интересует только параметр `REPLICATE` с операцией `Session#replicate()`. Все остальные операции `Session` имеют стандартизованные эквиваленты или альтернативы в `EntityManager` API, поэтому их можно не рассматривать.

Мы уже рассказали про параметры `PERSIST` и `REMOVE`. Давайте разберемся с транзитивным отсоединением, изменением и репликацией.

13.1.2. Транзитивное отсоединение и слияние

Предположим, что нужно извлечь из базы данных экземпляр `Item` и коллекцию `bids`, чтобы работать с ними в отсоединенном состоянии. В классе `Bid` эта связь отображается с помощью аннотации `@ManyToOne`. Она двунаправленная и имеет соответствующее отображение коллекции `@OneToMany` в классе `Item`:

Файл: /model/src/main/java/org/jpwh/model/filtering/cascade/Item.java

```
@Entity
public class Item {

    @OneToMany(
        mappedBy = "item",
        cascade = {CascadeType.DETACH, CascadeType.MERGE}
    )
    protected Set<Bid> bids = new HashSet<Bid>();

    // ...
}
```

Транзитивное отсоединение и слияние задаются при помощи параметров каскадирования `DETACH` и `MERGE`. Теперь загрузим экземпляр `Item` вместе с коллекцией `bids`:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(item.getBids().size(), 2);  ← Инициализация коллекции bids
em.detach(item);
```

`EntityManager#detach()` – каскадная операция; она исключает из контекста хранения экземпляр `Item`, а также коллекцию `bids`. Если коллекция `bids` не будет загружена, она не сможет стать отсоединенной (конечно, можно закрыть контекст хранения, фактически сделав отсоединенными *все* загруженные экземпляры сущностей).

В отсоединенном состоянии изменим значение `Item#name` и создадим новый объект `Bid`, связав его с экземпляром `Item`:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
item.setName("New Name");

Bid bid = new Bid(new BigDecimal("101.00"), item);
item.getBids().add(bid);
```

Работая с сущностями и коллекциями в отсоединенном состоянии, нужно обращать особое внимание на равенство и идентичность. Как объяснялось в разделе 10.3, мы должны переопределить методы `equals()` и `hashCode()` класса сущности `Bid`:

Файл: /model/src/main/java/org/jpwh/model/filtering/cascade/Bid.java

```
@Entity
public class Bid {

    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (!(other instanceof Bid)) return false;
        Bid that = (Bid) other;
```

```
        if (!this.getAmount().equals(that.getAmount()))
            return false;
        if (!this.getItem().getId().equals(that.getItem().getId()))
            return false;
        return true;
    }

    @Override
    public int hashCode() {
        int result = getAmount().hashCode();
        result = 31 * result + getItem().getId().hashCode();
        return result;
    }

    // ...
}
```

Два экземпляра `Bid` *равны*, если содержат одинаковую `цену` и связаны с одним и тем же экземпляром `Item`.

Закончив вносить изменения в отсоединенном состоянии, следующим шагом нужно их сохранить. Используя новый контекст хранения, выполним слияние отсоединенного экземпляра `Item`, чтобы Hibernate обнаружил изменения:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
Item mergedItem = em.merge(item);      ← ❶ Слияние объекта item
// select i.*, b.*
//  from ITEM i
//   left outer join BID b on i.ID = b.ITEM_ID
//  where i.ID = ?

for (Bid b : mergedItem.getBids()) {   ← ❷ Экземпляр Bid имеет значение идентификатора
    assertNotNull(b.getId());
}

em.flush();  ← ❸ Обнаружит изменение имени
// update ITEM set NAME = ? where ID = ?
// insert into BID values (?, ?, ?, ...)
```

❶ Hibernate выполнит слияние отсоединенного экземпляра `item`. Сначала он попытается найти экземпляр `Item` с заданным идентификатором в контексте хранения. В данном случае такого не окажется, поэтому объект `item` будет загружен из базы данных. Hibernate достаточно «умен», чтобы понять, что во время слияния также нужно загрузить коллекцию `bids`, поэтому он извлечет ее в том же запросе SQL. Затем скопирует значения свойств из отсоединенного объекта `item` в загруженный экземпляр, который возвращается в хранимом состоянии. Та же процедура применяется к каждому экземпляру `Bid`, и Hibernate обнаружит новый элемент в коллекции `bids`.

❷ Во время слияния Hibernate переведет новый экземпляр `Bid` в хранимое состояние. Теперь у него будет значение идентификатора.

❸ Выталкивая контекст хранения, Hibernate обнаружит, что свойство `name` объекта `Item` во время слияния изменилось. Новый экземпляр `Bid` также будет сохранен.

Каскадное слияние коллекций – мощная возможность. Представьте, как много кода пришлось бы написать для ее реализации в отсутствие Hibernate.

Немедленная загрузка коллекций во время слияния

В предыдущем примере мы сказали, что Hibernate достаточно «умен», чтобы загрузить коллекцию `Item#bids` при слиянии отсоединенного экземпляра `Item`. Во время слияния Hibernate всегда производит немедленную загрузку связей сущностей с помощью оператора `JOIN`, если связь определена с параметром `CascadeType.MERGE`. В предыдущем случае, когда коллекция `Item#bids` инициализировалась, отсоединялась и модифицировалась, это было «умным» поведением. Такая загрузка коллекции с помощью `JOIN` во время слияния является необходимой и оптимальной. Но при слиянии экземпляра `Item`, не имеющего инициализированной коллекции `bids` или прокси-объекта `seller`, Hibernate извлечет коллекцию и данные прокси-объекта с помощью `JOIN`. Операция слияния повлечет инициализацию этих связей в управляемом экземпляре `Item`, который будет возвращен. Параметр `CascadeType.MERGE` вынуждает Hibernate игнорировать и фактически переопределять любое отображение с параметром `FetchType.LAZY` (что допускается спецификацией JPA). В некоторых ситуациях такое поведение может быть неидеальным, но на момент написания книги его нельзя было настраивать.

Наш следующий пример будет менее запутанным: мы продемонстрируем каскадное обновление связанных сущностей.

13.1.3. Каскадное обновление

Класс сущности `User` (пользователь) связан отношением *один ко многим* с классом `BillingDetails` (платежные реквизиты): каждый пользователь приложения может иметь несколько кредитных карт, банковских счетов и т. д. Если вы не знакомы с классом `BillingDetails`, посмотрите отображения в главе 6.

Отношение между классами `User` и `BillingDetails` можно отобразить в виде однонаправленной ассоциации сущностей *один ко многим* (`@ManyToOne` отсутствует):

Файл: /model/src/main/java/org/jpwh/model/filtering/cascade/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @OneToMany(cascade = {CascadeType.PERSIST, CascadeType.REFRESH})
    @JoinColumn(name = "USER_ID", nullable = false)
    protected Set<BillingDetails> billingDetails = new HashSet<>();

    // ...
}
```

Для этой ассоциации указаны параметры каскадирования `PERSIST` и `REFRESH`. Параметр `PERSIST` упрощает сохранение платежных реквизитов; они становятся хранимыми при добавлении экземпляра `BillingDetails` в коллекцию хранимого экземпляра `User`.

В разделе 18.3 мы обсудим архитектуру, в которой контекст хранения остается открытым на протяжении долгого времени, что ведет к устареванию данных в управляемых экземплярах сущностей. Поэтому во время продолжительных диалоговых взаимодействий нужно повторно читать информацию из базы данных. Параметр каскадирования `REFRESH` гарантирует, что при перезагрузке экземпляра `User` Hibernate также обновит состояние каждого объекта `BillingDetails`, связанного с экземпляром `User`.

Файл: /examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
User user = em.find(User.class, USER_ID);          ← ❶ Загрузка экземпляра User

assertEquals(user.getBillingDetails().size(), 2);  ← ❷ Инициализация коллекции
for (BillingDetails bd : user.getBillingDetails()) {
    assertEquals(bd.getOwner(), "John Doe");
}

// Кто-то изменил платежные реквизиты в базе данных!

em.refresh(user);  ← ❸ Обновляет экземпляры BillingDetails
// select * from CREDITCARD join BILLINGDETAILS where ID = ?
// select * from BANKACCOUNT join BILLINGDETAILS where ID = ?
// select * from USERS
//  left outer join BILLINGDETAILS
//  left outer join CREDITCARD
//  left outer JOIN BANKACCOUNT
// where ID = ?

for (BillingDetails bd : user.getBillingDetails()) {
    assertEquals(bd.getOwner(), "Doe John");
}
```

❶ Экземпляр `User` загружается из базы данных.

❷ Отложенная коллекция `billingDetails` инициализируется в начале обхода элементов или при вызове `size()`.

❸ При изменении управляемого экземпляра `User` вызовом `refresh()` Hibernate каскадно выполнит эту операцию для всех управляемых экземпляров `BillingDetails` и обновит каждый из них с помощью SQL-выражения `SELECT`. Если ни одного из этих экземпляров не останется в базе данных, Hibernate возбудит исключение `EntityNotFoundException`. Затем Hibernate обновит экземпляр `User` и выполнит немедленную загрузку коллекции `billingDetails`, чтобы обнаружить новый экземпляр `BillingDetails`.

Это как раз тот случай, когда Hibernate действует не совсем оптимально. Сначала он выполнит SQL-выражение `SELECT` для каждого экземпляра `BillingDetails` из коллекции, а затем загрузит всю коллекцию снова, чтобы обнаружить добавленный экземпляр `BillingDetails`. Очевидно, что все это можно сделать одним выражением `SELECT`.

Последний способ каскадирования доступен только в Hibernate – это операция `replicate()`.

13.1.4. Каскадная репликация

Мы впервые столкнулись с репликацией в разделе 10.2.7. Эта нестандартная операция доступна в Hibernate-интерфейсе `Session`. Основным сценарием его использования является копирование данных из одной базы данных в другую.

Рассмотрим следующую связь *многие к одному* между классами `Item` и `User`:

Файл: /model/src/main/java/org/jpwh/model/filtering/cascade/Item.java

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "SELLER_ID", nullable = false)
    @org.hibernate.annotations.Cascade(
        org.hibernate.annotations.CascadeType.REPLICATE
    )
    protected User seller;

    // ...
}
```

Здесь в аннотации используется параметр каскадирования `REPLICATE`. Теперь загрузим экземпляр `Item` и его свойство `seller` из исходной базы данных:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);

assertNotNull(item.getSeller().getUsername());

tx.commit();
em.close();
```

Инициализация свойства Item#seller с отложенной загрузкой

Когда контекст хранения будет закрыт, экземпляры сущностей `Item` и `User` окажутся в отсоединенном состоянии. Теперь соединимся с другой базой данных и запишем отсоединенные данные:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Cascade.java

```
tx.begin();
EntityManager otherDatabase = // ... get EntityManager

otherDatabase.unwrap(Session.class)
    .replicate(item, ReplicationMode.OVERWRITE);
// select ID from ITEM where ID = ?
// select ID from USERS where ID = ?

tx.commit();
// update ITEM set NAME = ?, SELLER_ID = ?, ... where ID = ?
// update USERS set USERNAME = ?, ... where ID = ?
otherDatabase.close();
```

При вызове метода `replicate()` для отсоединенного экземпляра `Item` Hibernate выполнит SQL-выражения `SELECT`, чтобы выяснить, присутствует ли он и его свойство `seller` в базе данных. Затем, во время подтверждения транзакции, Hibernate запишет значения экземпляра `Item` и поля `seller` в целевую базу данных. В предыдущем примере эти строки уже были записаны, поэтому вы видите операцию `UPDATE` для каждой строки, перезаписывающую значения в базе данных. Если экземпляр `Item` или `User` отсутствует в целевой базе данных, будут выполнены две операции `INSERT`.

Последний способ каскадирования, который мы обсудим, – глобальная настройка, включающая каскадное сохранение всех связей сущностей.

13.1.5. Глобальное каскадное сохранение

Уровень хранения реализует стратегию *хранения по достижимости* (persistence by reachability), если любой экземпляр становится хранимым, когда приложение создает ссылку на этот экземпляр из уже хранимого экземпляра. В чистом виде стратегия хранения по достижимости реализуется посредством высокоуровневых или корневых объектов в базе данных, через которые достижимы все хранимые экземпляры. В идеальном случае, если до экземпляра нельзя добраться по ссылкам от корневого хранимого объекта, он должен перейти во временное состояние и быть удален из базы данных.

Ни Hibernate, ни любая другая реализация ORM не обеспечивают такого поведения. Фактически ни в одной базе данных SQL нет аналога корневого хранимого объекта, и ни один сборщик мусора, входящий в состав механизма хранения, не сможет определить экземпляров, на которые никто не ссылается. Объектно-ориентированные (сетевые) хранилища данных могут реализовывать алгоритмы сборки мусора, похожие на тот, который использует JVM для объектов в памяти; но в мире ORM этот вариант отсутствует, а сканирование всех таблиц для поиска записей, на которые никто не ссылается, неприемлемо с точки зрения производительности.

Тем не менее в идее хранения по достижимости есть определенные преимущества. Это позволяет переводить экземпляры из временного состояния в хранимое и передавать их в базу данных без многочисленных обращений к диспетчеру уровня хранения.

Активировать каскадное сохранение всех связей сущностей можно настройкой единицы хранения в файле метаданных отображения orm.xml:

Файл: /model/src/main/resources/filtering/DefaultCascadePersist.xml

```
<persistence-unit-metadata>
    <persistence-unit-defaults>
        <cascade-persist/>
    </persistence-unit-defaults>
</persistence-unit-metadata>
```

После этого ко всем связям сущностей в предметной модели, отображаемых этой единицей хранения, Hibernate будет применять параметр `CascadeType.PER-`

SIST. Каждый раз, когда будет создаваться ссылка из хранимого экземпляра сущности на временный, Hibernate автоматически будет переводить последний из временного состояния в хранимое.

Фактически параметры каскадирования – это предопределенные реакции механизма хранения на события жизненного цикла. Если вам потребуется выполнить свою процедуру при сохранении или загрузке данных, можно определить собственные приемники и обработчики событий.

13.2. Прием и обработка событий

В этом разделе мы рассмотрим три различных API для работы с приемниками и обработчиками событий жизненного цикла хранилища, доступные в JPA и Hibernate. Они позволяют:

- использовать стандартные методы обратного вызова и приемники событий жизненного цикла, которые определены в JPA;
- определить реализацию интерфейса `org.hibernate.Interceptor` и использовать ее при работе с экземпляром `Session`;
- использовать механизм расширения Hibernate через интерфейс `org.hibernate.event`.

Начнем со стандартных методов обратного вызова JPA. Они дают удобный доступ к событиям сохранения, загрузки и удаления.

13.2.1. Приемники событий JPA и обратные вызовы

Предположим, что при сохранении нового экземпляра сущности требуется послать электронное письмо системному администратору. Для этого, во-первых, нужно создать приемник событий жизненного цикла с методом обратного вызова, отмеченного аннотацией `@PostPersist`, как показано в следующем листинге.

Листинг 13.1 ❖ Уведомление администратора о сохранении нового экземпляра сущности

Файл: /model/src/main/java/org/jpwh/model/filtering/callback/PersistEntityListener.java

```
public class PersistEntityListener {

    @PostPersist
    public void notifyAdmin(Object entityInstance) {

        User currentUser = CurrentUser.INSTANCE.get();
        Mail mail = Mail.INSTANCE;

        mail.send(
            "Entity instance persisted by "
                + currentUser.getUsername()
                + ": "
                + entityInstance
        );
    }
}
```

❶ Конструктор приемника событий жизненного цикла сущностей

❷ Аннотация превращает метод notfyAdmin() в метод обратного вызова

❸ Получение информации о пользователе и отправка почты

❶ Класс приемника событий жизненного цикла сущностей не должен иметь конструктора, или конструктор должен быть общедоступным и не иметь аргументов. От класса не требуется реализовать какой-то специальный интерфейс. Класс приемника событий не имеет состояния, а его экземпляры автоматически создаются и уничтожаются реализацией JPA.

❷ Любой метод класса приемника событий можно отметить как метод обратного вызова. Метод `notifyAdmin()` в данном примере будет вызываться после сохранения нового экземпляра сущности в базу данных.

❸ Поскольку классы приемников событий не имеют состояния, получить более содержательную информацию может быть затруднительно. Здесь нам требуется получить экземпляр текущего авторизованного пользователя и обратиться к почтовой системе для отправки уведомления. Простейшим решением будет использование локальных переменных потока и объектов-одиночек; вы можете найти исходные тексты классов `CurrentUser` и `Mail` в коде примеров.

Метод обратного вызова принимает единственный параметр типа `Object`: экземпляр сущности, меняющий состояние. Если понадобится определить метод обратного вызова для конкретного типа сущности, укажите конкретный тип параметра. Метод обратного вызова может иметь любой модификатор доступа – он не обязан быть общедоступным. Он не должен быть ни статическим, ни финальным и не должен ничего не возвращать. Если метод обратного вызова возбудит неконтролируемое исключение `RuntimeException`, Hibernate отменит выполняемую операцию и отметит текущую транзакцию как подлежащую откату. Если метод обратного вызова объявит и возбудит контролируемое исключение `Exception`, Hibernate завернет его в неконтролируемое исключение `RuntimeException`.

Внедрение зависимостей в классы приемников событий

Часто при реализации приемника событий требуется получить доступ к контекстной информации и обращаться к API. В предыдущем примере понадобилось получить текущего зарегистрированного пользователя и обратиться к API для отправки почты. Простейшего решения на основе локальных переменных потока и объектов-одиночек может оказаться недостаточно для больших и сложных приложений. JPA также стандартизирует интеграцию при помощи CDI, поэтому класс приемника событий может использовать механизм внедрения зависимостей посредством аннотации `@Inject`. При обращении к классу приемника контейнер CDI автоматически внедрит контекстную информацию. Но обратите внимание, что даже при использовании CDI нельзя внедрить экземпляр `EntityManager` в приемник событий для доступа к базе данных. Далее в этой главе мы обсудим другой способ доступа к базе данных из приемника событий (Hibernate).

Каждую аннотацию для методов обратного вызова можно использовать в определении приемника событий только один раз, т. е. лишь один метод может быть отмечен аннотацией `@PostPersist`. Все доступные аннотации для методов обратных вызовов перечислены в табл. 13.2.

Таблица 13.2. Аннотации методов обратного вызова для событий жизненного цикла

Аннотация	Описание
`@PostLoad`	Метод вызывается после загрузки экземпляра сущности в контекст хранения, либо в результате поиска по идентификатору, в ходе навигации и инициализации прокси-объекта/коллекции, либо при выполнении запроса. Метод также вызывается после изменения уже хранимого экземпляра
`@PrePersist`	Вызывается сразу после вызова операции `persist()` для экземпляра сущности. Также вызывается после выполнения операции `merge()` для временной сущности, когда ее временное состояние будет скопировано в хранимый экземпляр сущности. Метод также будет вызван для связей сущностей с опцией `CascadeType.PERSIST`
`@PostPersist`	Вызывается после сохранения экземпляра сущности в базе данных и присваивания ему значения идентификатора. Это может произойти либо после операций `persist()` или `merge()`, либо позже, во время выталкивания контекста хранения, если генератор идентификаторов срабатывает перед вставкой (см. раздел 4.2.5). Метод также будет вызван для связей сущностей с параметром `CascadeType.PERSIST`
`@PreUpdate`, `@PostUpdate`	Методы вызываются до и после синхронизации контекста хранения с базой данных, т. е. до и после выталкивания контекста, но только в том случае, когда требуется синхронизация сущности (если она определена как изменившаяся)
`@PreRemove`, `@PostRemove`	Вызываются при выполнении `remove()` или каскадном удалении экземпляра сущности, а также после удаления записи из базы данных, когда происходит выталкивание контекста хранения

Чтобы организовать прием событий, возникающих в конкретной сущности, необходимо явно указать класс приемника событий перед объявлением класса сущности, например:

Файл: /model/src/main/java/org/jpwh/model/filtering/callback/Item.java

```
@Entity
@EntityListeners(
    PersistEntityListener.class
)
public class Item {

    // ...
}
```

Аннотация `@EntityListeners` может принимать массив классов приемников событий, если их несколько. Если в нескольких приемниках определены методы обратного вызова для одних и тех же событий, Hibernate будет вызывать их в порядке перечисления. Связать классы приемников с сущностями можно также в XML-файле метаданных при помощи дочерних элементов `<entity-listener>` в элементах `<entity>`.

Чтобы организовать обработку отдельных событий жизненного цикла, не обязательно создавать отдельный класс приемника. Например, можно определить метод `notifyAdmin()` в классе сущности `User`:

Файл: /model/src/main/java/org/jpwh/model/filtering/callback/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @PostPersist
    public void notifyAdmin(){
        User currentUser = CurrentUser.INSTANCE.get();
        Mail mail = Mail.INSTANCE;
        mail.send(
        "Entity instance persisted by "
                + currentUser.getUsername()
                + ": "
                + this
        );
    }

    // ...
}
```

Обратите внимание, что методы обратного вызова в классе сущности не имеют аргументов: текущая сущность, меняющая состояние, доступна через переменную `this`. Не допускается определять в одном классе несколько методов обратного вызова для одного и того же события. Но одно и то же событие можно обрабатывать в нескольких методах обратного вызова в разных классах сущностей или с помощью приемника событий и в самом классе сущности.

Также есть возможность определить методы обратного вызова для целой иерархии классов в суперклассах сущностей. Если в конкретном подклассе сущности потребуется отключить методы обратного вызова суперкласса, достаточно отметить подкласс аннотацией `@ExcludeSuperclassListeners` или отобразить его в метаданных XML с элементом `<exclude-superclass-listeners>`.

Используя метаданные XML, можно определить классы приемников событий, применяемые по умолчанию ко всем сущностям в единице хранения:

Файл: /model/src/main/resources/filtering/EventListeners.xml

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener
        class="org.jpwh.model.filtering.callback.PersistEntityListener"/>
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

Если потребуется отключить приемник событий по умолчанию в конкретной сущности, его следует отобразить с элементом `<exclude-default-listeners>` в файле метаданных XML или отметить аннотацией `@ExcludeDefaultListeners`:

Файл: /model/src/main/java/org/jpwh/model/filtering/callback/User.java

```
@Entity
@Table(name = "USERS")
@ExcludeDefaultListeners
public class User {

    // ...
}
```

Но помните, что вызываться будут все подключенные приемники событий. Если подключить/связать приемники событий в метаданных XML и аннотациях, Hibernate вызовет их в следующем порядке:

1) приемники событий по умолчанию из единицы хранения, в порядке их перечисления в метаданных XML;
2) приемники событий, связанные с сущностью при помощи аннотации `@EntityListeners`, в порядке перечисления;
3) сначала методы обратного вызова, объявленные в суперклассах, начиная с самого общего. Последними – методы обратного вызова в классе сущности.

Приемники событий JPA и методы обратного вызова – это простейший способ вызова процедур при наступлении определенных событий жизненного цикла. В качестве альтернативы в Hibernate имеется более гибкий и мощный API: `org.hibernate.Interceptor`.

13.2.2. Реализация перехватчиков Hibernate

Предположим, что требуется вносить записи о модификации данных в журнал аудита в отдельной таблице базы данных, например с информацией о создании и изменении каждого экземпляра `Item`. В журнал аудита должны записываться имя пользователя, дата, время и тип события, а также идентификатор измененного экземпляра `Item`.

Журналы аудита, как правило, ведутся при помощи триггеров в базе данных. С другой стороны, иногда лучше делать это в приложении, особенно когда требуется обеспечить переносимость между несколькими базами данных.

Для реализации журнала аудита потребуется несколько ингредиентов. Во-первых, нужно отметить все классы сущностей, для которых требуется вносить записи в журнал аудита. Затем определить журналируемые данные, такие как имя пользователя, дата, время и тип изменения. И наконец, связать все воедино с помощью `org.hibernate.Interceptor`, который будет автоматически создавать записи аудита.

Сначала создадим интерфейс-маркер `Auditable`:

Файл: /model/src/main/java/org/jpwh/model/filtering/interceptor/Auditable.java

```
public interface Auditable {
    public Long getId();
}
```

Этот интерфейс требует, чтобы класс хранимой сущности открывал доступ к идентификатору посредством метода чтения; это свойство понадобится для записи в журнал аудита. После этого внесение записей о конкретном хранимом классе в журнал аудита становится тривиальным. Нужно добавить его в объявление класса, как, например, в классе `Item`:

Файл: /model/src/main/java/org/jpwh/model/filtering/interceptor/Item.java

```
@Entity
public class Item implements Auditable {

    // ...
}
```

Далее нужно создать новый класс хранимой сущности `AuditLogRecord`, определив в нем всю информацию для записи в таблицу журнала аудита:

Файл: /model/src/main/java/org/jpwh/model/filtering/interceptor/AuditLogRecord.java

```
@Entity
public class AuditLogRecord {

    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    protected Long id;

    @NotNull
    protected String message;

    @NotNull
    protected Long entityId;

    @NotNull
    protected Class entityClass;

    @NotNull
    protected Long userId;

    @NotNull
    @Temporal(TemporalType.TIMESTAMP)
    protected Date createdOn = new Date();

    // ...
}
```

Нам нужно организовать сохранение экземпляра `AuditLogRecord`, когда Hibernate сохраняет или обновляет экземпляр `Item` в базе данных. Перехватчик Hibernate может делать это автоматически. Но вместо реализации всех методов интерфейса `org.hibernate.Interceptor` достаточно лишь расширить класс `EmptyInterceptor` и переопределить нужные методы, как показано далее.

Листинг 13.2 ❖ Перехватчик Hibernate, записывающий в журнал события изменения

Файл: /examples/src/test/java/org/jpwh/test/filtering/AuditLogInterceptor.java

```
public class AuditLogInterceptor extends EmptyInterceptor {

    protected Session currentSession;  ← ❶ Обеспечивает доступ к базе данных
    protected Long currentUserId;
    protected Set<Auditable> inserts = new HashSet<Auditable>();
    protected Set<Auditable> updates = new HashSet<Auditable>();

    public void setCurrentSession(Session session) {
        this.currentSession = session;
    }

    public void setCurrentUserId(Long currentUserId) {
        this.currentUserId = currentUserId;
    }

    public boolean onSave(Object entity, Serializable id,      ← ❷ Вызывается, когда
                          Object[] state, String[] propertyNames,   экземпляр становится
                          Type[] types)                             хранимым
        throws CallbackException {

        if (entity instanceof Auditable)
            inserts.add((Auditable)entity);
        return false;  ← Мы не изменяли состояния
    }

    public boolean onFlushDirty(Object entity, Serializable id,  ← ❸ Вызывается
                                Object[] currentState,               при изменении
                                Object[] previousState,              состояния сущности
                                String[] propertyNames, Type[] types)
        throws CallbackException {

        if (entity instanceof Auditable)
            updates.add((Auditable)entity);
        return false;  ← Мы не меняли текущего состояния
    }

    // ...
}
```

❶ Нам потребуется доступ к базе данных для записи в журнал аудита, поэтому мы добавили в перехватчик Hibernate объект `Session`. Также нам понадобится сохранять в каждой записи журнала аудита идентификатор текущего авторизованного пользователя. Переменные экземпляра `inserts` и `updates` – это коллекции, в которых перехватчик сохраняет свое состояние.

❷ Этот метод вызывается, когда экземпляр сущности становится хранимым.

❸ Этот метод вызывается, если во время выталкивания контекста хранения в экземпляре сущности будут обнаружены изменения.

Перехватчик сохраняет все измененные экземпляры `Auditable` в коллекциях `inserts` и `updates`. Обратите внимание, что в методе `onSave()` экземпляр сущности может не иметь идентификатора. Hibernate гарантирует присваивание идентификатора во время выталкивания контекста, поэтому запись в журнал аудита выполняется в методе обратного вызова `postFlush()`, не показанном в листинге 13.2:

Файл: /examples/src/test/java/org/jpwh/test/filtering/AuditLogInterceptor.java

```
public class AuditLogInterceptor extends EmptyInterceptor {

    // ...

    public void postFlush(Iterator iterator) throws CallbackException {   ❶ Выполняет запись в журнал аудита

        Session tempSession =                                              ❷ Создает временный объект Session
            currentSession.sessionWithOptions()
                .transactionContext()
                .connection()
                .openSession();
        try {
            for (Auditable entity : inserts) {                             ❸ Сохраняет экземпляры AuditLogRecords
                tempSession.persist(
                    new AuditLogRecord("insert", entity, currentUserId)
                );
            }
            for (Auditable entity : updates) {
                tempSession.persist(
                    new AuditLogRecord("update", entity, currentUserId)
                );
            }

            tempSession.flush();                                           ❹ Закрывает временный сеанс
        } finally {
            tempSession.close();
            inserts.clear();
            updates.clear();
        }
    }
}
```

❶ Этот метод будет вызываться после выталкивания контекста хранения. Здесь мы записываем данные в журнал аудита для всех операций добавления и изменения, которые мы запомнили ранее.

❷ Мы не можем получить доступа к оригинальному контексту хранения – объекту `Session`, вызывающему данный перехватчик. Во время вызова перехватчиков сеанс (`Session`) находится в уязвимом состоянии. Однако Hibernate дает возможность создать новый экземпляр `Session`, наследующий некоторую информацию из оригинального экземпляра `Session`, с помощью метода `sessionWithOptions()`. Новый временный экземпляр `Session` работает в той же транзакции и с тем же соединением с базой данных, что и оригинальный экземпляр `Session`.

❸ С помощью объекта Session для каждой операции добавления и изменения создается новый экземпляр AuditLogRecord.
❹ Выталкиваем контекст и закрываем временный сеанс (Session) независимо от оригинального сеанса (Session).

Теперь можно подключать этот перехватчик во время создания объекта EntityManager, как свойство Hibernate:

Файл: /examples/src/test/java/org/jpwh/test/filtering/AuditLogging.java

```
EntityManagerFactory emf = JPA.getEntityManagerFactory();

Map<String, String> properties = new HashMap<String, String>();
properties.put(
    org.hibernate.jpa.AvailableSettings.SESSION_INTERCEPTOR,
    AuditLogInterceptor.class.getName()
);

EntityManager em = emf.createEntityManager(properties);
```

Определение перехватчиков по умолчанию

Если для каждого экземпляра EntityManager потребуется использовать перехватчик по умолчанию, в параметре hibernate.ejb.interceptor (в файле persistence.xml) нужно указать имя класса, реализующего интерфейс org.hibernate.Interceptor. Обратите особое внимание, что, в отличие от перехватчика уровня сеанса, Hibernate разделяет этот перехватчик по умолчанию между потоками, поэтому он должен быть потокобезопасным! Реализация AuditLogInterceptor из примера *не* является потокобезопасной.

Теперь перехватчик AuditLogInterceptor подключен к экземпляру EntityManager, но его еще нужно настроить, передав экземпляр Session и идентификатор текущего авторизованного пользователя. Это требует нескольких приведений типов для доступа к Hibernate API:

Файл: /examples/src/test/java/org/jpwh/test/filtering/AuditLogging.java

```
Session session = em.unwrap(Session.class);
AuditLogInterceptor interceptor =
    (AuditLogInterceptor) ((SessionImplementor) session).getInterceptor();
interceptor.setCurrentSession(session);
interceptor.setCurrentUserId(CURRENT_USER_ID);
```

Экземпляр EntityManager теперь готов к использованию, и при каждом сохранении и изменении экземпляра Item с его помощью в журнал аудита будет вноситься запись.

Перехватчики Hibernate более гибкие, чем приемники событий JPA и методы обратного вызова, – они позволяют получить гораздо больше контекстной информации при наступлении события. Наряду с этим Hibernate позволяет еще глубже внедряться в свое ядро благодаря своей полностью расширяемой системе событий.

13.2.3. Базовый механизм событий

Ядро механизма Hibernate основано на модели событий и приемников. Например, если Hibernate потребуется сохранить экземпляр сущности, он сгенерирует событие. И все, кто принимает события этого типа, сможет среагировать на сохранение данных. То есть вся базовая функциональность Hibernate реализована в виде набора приемников по умолчанию, обрабатывающих все события Hibernate.

Hibernate имеет открытую архитектуру: вы можете создавать и использовать собственные приемники событий. Можно подменять приемники по умолчанию или дополнять их, чтобы получить побочные эффекты или выполнить дополнительные процедуры. Подмена приемников событий происходит редко; это означало бы, что ваша реализация берет на себя часть базовой функциональности Hibernate.

Фактически каждый метод интерфейса `Session` (и его более простого собрата, интерфейса `EntityManager`) соответствует некоторому событию. Методы `find()` и `load()` генерируют событие `LoadEvent`, и по умолчанию это событие обрабатывается экземпляром `DefaultLoadEventListener`.

Ваш собственный приемник должен реализовать подходящий интерфейс для события, которое он будет обрабатывать, и/или расширять один из вспомогательных базовых классов Hibernate или приемников событий по умолчанию. Далее показан пример реализации собственного приемника событий загрузки.

Листинг 13.3 ❖ Собственный слушатель событий загрузки

Файл: /examples/src/test/java/org/jpwh/test/filtering/SecurityLoadListener.java

```
public class SecurityLoadListener extends DefaultLoadEventListener {

    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
        throws HibernateException {

        boolean authorized =
            MySecurity.isAuthorized(
                event.getEntityClassName(), event.getEntityId()
            );

        if (!authorized)
            throw new MySecurityException("Unauthorized access");

        super.onLoad(event, loadType);
    }

}
```

Этот приемник может выполнять нестандартный код авторизации. Фактически приемник должен быть объектом-одиночкой (singleton); он совместно используется разными контекстами хранения, поэтому в нем не должно быть переменных состояния, связанных с транзакциями. Список всех оригинальных событий и интерфейсов приемников событий в Hibernate можно найти в документации Javadoc для пакета `org.hibernate.event`.

Приемники для каждого базового события можно настроить в файле persistence.xml в элементе `<persistenceunit>`:

Файл: /model/src/main/resources/META-INF/persistence.xml

```
<properties>
    <property name="hibernate.ejb.event.load"
              value="org.jpwh.test.filtering.SecurityLoadListener"/>
</properties>
```

Название свойства конфигурации всегда начинается с префикса `hibernate.ejb.event`, за которым указывается тип принимаемого события. Список всех типов событий находится в классе `org.hibernate.event.spi.EventType`. В качестве значения свойства можно указать список имен классов-приемников, перечисленных через запятую; Hibernate будет вызывать их в порядке перечисления.

Но расширять систему базовых событий Hibernate собственной функциональностью требуется нечасто. Для большинства ситуаций достаточно интерфейса `org.hibernate.Interceptor`. Он позволяет получить больше возможностей и заместить любую часть ядра механизма Hibernate согласно принципу модульности.

Реализация внесения записей в журнал аудита, показанная в предыдущем разделе, была очень простой. Если для аудита потребуется сохранять больше информации, например значения измененных полей сущности, обратите внимание на *Hibernate Envers*.

13.3. Аудит и версионирование с помощью Hibernate Envers

Envers – это проект семейства Hibernate, задачей которого являются ведение журнала аудита и хранение нескольких версий данных в базе. Своим действием он напоминает системы управления версиями, такие как Subversion или Git, которые могут быть вам уже знакомы.

После настройки механизма Envers он автоматически будет сохранять в отдельных таблицах копии при добавлении, изменении или удалении данных из основных таблиц. Envers использует программный интерфейс событий Hibernate, который вы видели в предыдущем разделе. Envers перехватывает события и, когда Hibernate сохраняет изменения в базу данных, создает копии данных и сохраняет их в собственных таблицах.

Envers группирует все изменения, произведенные в рамках единицы работы, то есть в транзакции, в один набор изменений с общей версией. Для извлечения архивных данных с заданной версией или меткой времени можно создавать запросы, используя интерфейс Envers API: например, «найти все экземпляры `Item` по состоянию на прошлую пятницу». Но для начала следует подключить Envers к приложению.

13.3.1. Включение ведения журнала аудита

Как только JAR-файл проекта Envers будет добавлен в путь поиска классов (или, как показано в коде примеров, в зависимости Maven), его можно использовать без

дополнительных настроек. Аудит класса сущности можно осуществлять избирательно, при помощи аннотации `@org.hibernate.envers.Audited`.

Листинг 13.4 ❖ Включение аудита для сущности `Item`

Файл: /model/src/main/java/org/jpwh/model/filtering/envers/Item.java

```
@Entity
@org.hibernate.envers.Audited
public class Item {

    @NotNull
    protected String name;

    @OneToMany(mappedBy = "item")
    @org.hibernate.envers.NotAudited
    protected Set<Bid> bids = new HashSet<Bid>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "SELLER_ID", nullable = false)
    protected User seller;

    // ...
}
```

Теперь приложение будет вести журнал аудита для экземпляров `Item` и всех свойств сущности. Чтобы выключить аудит для конкретного свойства, нужно отметить его аннотацией `@NotAudited`. В данном случае Envers будет игнорировать коллекцию `bids`, но осуществлять аудит для `seller`. Также требуется поместить аннотацию `@Audited` перед классом `User`.

Hibernate сгенерирует (или будет искать) дополнительные таблицы в базе для хранения архивных данных каждого экземпляра `Item` и `User`. Схема таблиц представлена на рис. 13.1.

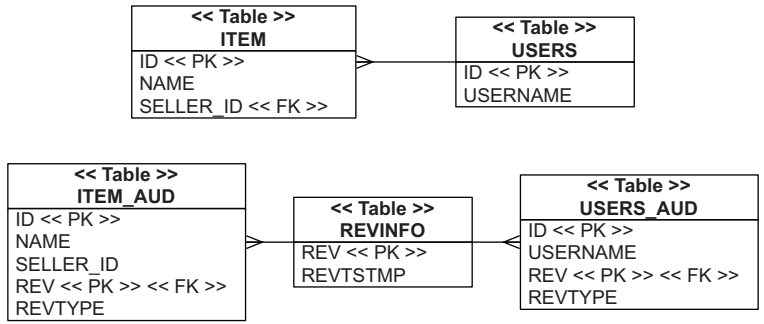


Рис. 13.1 ❖ Таблицы журнала аудита для сущностей `Item` и `User`

История изменений хранится в таблицах `ITEM_AUD` и `USERS_AUD`, соответствующих экземплярам `Item` и `User`. При изменении данных с последующим подтверждением транзакции Hibernate сохранит новый номер версии с меткой времени

в таблицу REVINFO. Затем для всех модифицированных экземпляров сущностей из набора изменений, для которых настроен аудит, создаст копию данных и сохранит в таблицы журнала аудита. Внешние ключи в виде номеров версий связывают набор изменений воедино. В столбце REVTYPE хранится тип изменения: был ли конкретный экземпляр сущности добавлен, изменен или удален во время транзакции. Envers никогда не удаляет информацию о версиях и архивные данные; даже после вызова метода remove() для экземпляра Item его предыдущая версия по-прежнему будет храниться в таблице ITEM_AUD.

Давайте выполним несколько транзакций, чтобы понять, как это работает.

13.3.2. Ведение аудита

Далее мы рассмотрим несколько транзакций, использующих экземпляр Item и его поле seller типа User. Мы создадим и сохраним экземпляры Item и User, затем изменим их и, наконец, удалим экземпляр Item.

Этот код уже должен быть вам знаком. Когда вы будете работать с объектом EntityManager, Envers автоматически создаст записи в журнале аудита:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

User user = new User("johndoe");
em.persist(user);

Item item = new Item("Foo", user);
em.persist(item);

tx.commit();
em.close();
```

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
item.setName("Bar");
item.getSeller().setUsername("doejohn");

tx.commit();
em.close();
```

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
em.remove(item);

tx.commit();
em.close();
```

Во время выполнения этих транзакций Envers создаст записи в журнале аудита в виде трех наборов изменений. Для доступа к этим архивным данным для начала нужно узнать версию, соответствующую требуемому набору изменений.

13.3.3. Поиск версий

Используя интерфейс Envers `AuditReader`, можно отыскать номер версии для каждого набора изменений:

Листинг 13.5 ❖ Получение номеров версий для наборов изменений

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
AuditReader auditReader = AuditReaderFactory.get(em);   ← ❶ Интерфейс AuditReader
Number revisionCreate =                                 ← ❷ Получение версии номера
       auditReader.getRevisionNumberForDate(TIMESTAMP_CREATE);
Number revisionUpdate =
       auditReader.getRevisionNumberForDate(TIMESTAMP_UPDATE);
Number revisionDelete =
       auditReader.getRevisionNumberForDate(TIMESTAMP_DELETE);

List<Number> itemRevisions = auditReader.getRevisions(Item.class, ITEM_ID);
assertEquals(itemRevisions.size(), 3);                  ← ❸ Поиск наборов изменений
for (Number itemRevision : itemRevisions) {
    Date itemRevisionTimestamp = auditReader.getRevisionDate(itemRevision);
    // ...                                              ← ❹ Получение метки времени
}

List<Number> userRevisions = auditReader.getRevisions(User.class, USER_ID);
assertEquals(userRevisions.size(), 2);                  ← ❺ Подсчет количества версий
```

❶ Интерфейс `AuditReader` – это основа Envers API. Получить доступ к нему можно с помощью объекта `EntityManager`.

❷ Получив метку времени, можно отыскать номер версии набора изменений, выполненных ранее этой метки или одновременно с ней.

❸ Если метка времени отсутствует, можно извлечь все номера версий, соответствующие конкретному экземпляру сущности, для которой настроен аудит. Эта операция отыщет все наборы изменений, в которых заданный экземпляр `Item` был создан, изменен или удален. В нашем примере мы создали, изменили и удалили экземпляр `Item`. Как результат мы получили три версии.

❹ Имея номер версии, можно узнать метку времени, когда Envers зарегистрировал набор изменений.

❺ Мы создали и изменили экземпляр `User`, поэтому в результате мы получили только две версии.

В листинге 13.5 предполагается, что у нас есть метка времени (приблизительная) транзакции или идентификатор сущности (для получения всех версий). Если эта информация отсутствует, придется исследовать журнал аудита с помощью запросов. Это бывает также полезно, когда нужно показать все наборы изменений в интерфейсе пользователя приложения.

Следующий код извлекает все версии сущности `Item` и загружает каждую версию `Item` вместе с данными аудита для конкретного набора изменений:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
AuditQuery query = auditReader.createQuery()      ← ❶ Запрос для получения данных аудита
    .forRevisionsOfEntity(Item.class, false, false);

List<Object[]> result = query.getResultList();   ← ❷ Получение данных аудита
for (Object[] tuple : result) {

    Item item = (Item) tuple[0];   ← ❸ Получение информации о версии
    DefaultRevisionEntity revision = (DefaultRevisionEntity)tuple[1];
    RevisionType revisionType = (RevisionType)tuple[2];

    if (revision.getId() == 1) {   ← ❹ Получение типа версии
        assertEquals(revisionType, RevisionType.ADD);
        assertEquals(item.getName(), "Foo");
    } else if (revision.getId() == 2) {
        assertEquals(revisionType, RevisionType.MOD);
        assertEquals(item.getName(), "Bar");
    } else if (revision.getId() == 3) {
        assertEquals(revisionType, RevisionType.DEL);
        assertNull(item);
    }
}
```

❶ Если не известны ни метки времени, ни номера версий, можно создать запрос с помощью метода `forRevisionsOfEntity()` и получить все записи из журнала аудита для конкретной сущности.

❷ Этот запрос возвращает записи из журнала в виде коллекции `List` с элементами типа `Object[]`.

❸ Каждый кортеж в списке результатов содержит экземпляр сущности для конкретной версии, информацию о версии (включая номер версии и метку времени), а также тип версии.

❹ Тип версии указывает на причину ее создания: вставка, изменение или удаление из базы данных.

Номера версий увеличиваются последовательно; больший номер соответствует более поздней версии экземпляра сущности. Теперь у нас есть номера версий для трех наборов изменений в журнале аудита, и мы можем получить доступ к архивным данным.

13.3.4. Получение архивных данных

Зная номера версий, можно извлекать различные состояния товара (`Item`) и его продавца (`seller`).

Листинг 13.6 ❖ Загрузка предыдущих версий экземпляра сущности

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
Item item = auditReader.find(Item.class, ITEM_ID, revisionCreate);
assertEquals(item.getName(), "Foo");
assertEquals(item.getSeller().getUsername(), "johndoe");

Item modifiedItem = auditReader.find(Item.class,
    ITEM_ID, revisionUpdate);
assertEquals(modifiedItem.getName(), "Bar");
assertEquals(modifiedItem.getSeller().getUsername(), "doejohn");

Item deletedItem = auditReader.find(Item.class,
    ITEM_ID, revisionDelete);
assertNull(deletedItem);

User user = auditReader.find(User.class,
    USER_ID, revisionDelete);
assertEquals(user.getUsername(), "doejohn");
```

❶ Возвращает экземпляр, для которого настроен аудит

❷ Загрузка измененного экземпляра Item

❸ Работа с удаленным экземпляром Item

❹ Получение самой последней версии

❶ Если передать методу `find()` конкретную версию, он вернет соответствующее состояние экземпляра сущности. Эта операция возвращает состояние экземпляра `Item` после его создания.

❷ Эта операция возвращает состояние экземпляра `Item` после его изменения. Обратите внимание, что значение свойства `seller` для этого набора изменений извлекается автоматически.

❸ Эта версия соответствует удаленному экземпляру `Item`, поэтому `find()` вернет `null`.

❹ В этой версии экземпляр `User` остался неизменным, поэтому Envers возвращает наиболее близкое к данной версии состояние.

Операция `AuditReader#find()` возвращает только один экземпляр сущности – так же, как и `EntityManager#find()`. Но возвращаемые экземпляры сущностей *не* находятся в хранимом состоянии – они не управляются контекстом хранения. При изменении старой версии экземпляра `Item` Hibernate не выполнит обновления базы данных. Обращайтесь с экземпляром сущности, возвращаемым интерфейсом `AuditReader` API, как с отсоединенным или доступным только для чтения.

Кроме того, `AuditReader` имеет API, похожий на `Criteria` API в Hibernate, позволяющий выполнять произвольные запросы (см. раздел 16.3).

Листинг 13.7 ❖ Получение старых версий экземпляров сущностей

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
AuditQuery query = auditReader.createQuery()
    .forEntitiesAtRevision(Item.class, revisionUpdate);

query.add(
    AuditEntity.property("name").like("Ba", MatchMode.START)
);
```

❶ Возвращает экземпляры Item, соответствующие конкретной версии

❷ Добавление ограничения

```
query.add(   ← ❸ Добавление ограничения
    AuditEntity.relatedId("seller").eq(USER_ID)
);

query.addOrder(   ← ❹ Упорядочение результатов
    AuditEntity.property("name").desc()
);

query.setFirstResult(0);   ← ❺ Постраничный вывод
query.setMaxResults(10);

assertEquals(query.getResultList().size(), 1);
Item result = (Item)query.getResultList().get(0);
assertEquals(result.getSeller().getUsername(), "doejohn");
```

❶ Этот запрос вернет экземпляры `Item`, соответствующие конкретной версии и набору изменений.
❷ Вы можете добавлять в запрос ограничения: здесь значение поля `Item#name` должно начинаться с «Ba».
❸ Ограничения могут касаться связей сущностей: например, можно отыскать версию товара (`Item`), проданного конкретным пользователем (`User`).
❹ Результаты запроса можно упорядочивать.
❺ Объемные результаты можно выводить постранично.

Также Envers поддерживает операцию проекции. Следующий запрос вернет только поле `Item#name`, соответствующее конкретной версии:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
AuditQuery query = auditReader.createQuery()
    .forEntitiesAtRevision(Item.class, revisionUpdate);

query.addProjection(
    AuditEntity.property("name")
);

assertEquals(query.getResultList().size(), 1);
String result = (String)query.getSingleResult();
assertEquals(result, "Bar");
```

Наконец, вам может понадобиться откатить экземпляр сущности к предыдущей версии. Это можно сделать, вызвав метод `Session#replicate()` и изменив существующую запись в базе данных. Следующий пример загружает экземпляр `User` из первого набора изменений, а затем заменяет актуальный экземпляр `User` в базе данных предыдущей версией:

Файл: /examples/src/test/java/org/jpwh/test/filtering/Envers.java

```
User user = auditReader.find(User.class, USER_ID, revisionCreate);

em.unwrap(Session.class)
    .replicate(user, ReplicationMode.OVERWRITE);
em.flush();
```

```
em.clear();

user = em.find(User.class, USER_ID);
assertEquals(user.getUsername(), "johndoe");
```

Envers зарегистрирует это изменение в журнале аудита как обновление; это лишь еще одна версия экземпляра `User`.

Хронологические данные – это сложная тема, и мы рекомендуем изучить документацию Envers для получения дополнительной информации. Добавить в журнал аудита такую информацию, как имя пользователя, сделавшего изменение, не так уж сложно. В документации также рассказывается о выборе различных стратегий отслеживания изменений и модификации схемы базы данных, используемой Envers.

Теперь представьте, что вам не требуется видеть все данные в базе. Например, у текущего пользователя приложения может не быть прав для просмотра всей информации.

Как правило, для отбора требуемой информации в запросы добавляются различные условия. Но это довольно сложно при работе с такой областью, как безопасность, поскольку требуется менять большинство запросов в приложении. Но с помощью динамических фильтров данных Hibernate можно изолировать эти ограничения в одном месте.

13.4. Динамическая фильтрация данных

Первой областью применения динамической фильтрации является обеспечение безопасности данных. В приложении CaveatEmptor пользователю (`User`) можно присвоить `ранг` в виде целого числа:

Файл: /model/src/main/java/org/jpwh/model/filtering/dynamic/User.java

```
@Entity
@Table(name = "USERS")
public class User {

    @NotNull
    protected int rank = 0;

    // ...
}
```

Теперь предположим, что пользователи могут делать ставки только для тех товаров, которые предлагают пользователи с равным или меньшим рангом. Говоря на языке бизнеса, имеется несколько групп пользователей, определяемых произвольно выбранным рангом (числом), и пользователи могут торговать только с теми, у кого ранг такой же или меньше.

Для реализации этого требования нам пришлось бы переписать все запросы, загружающие экземпляры `Item` из базы данных. Нужно было бы проверять ранг у объекта `Item#seller` и сравнивать его с рангом текущего пользователя. Hibernate может сделать эту работу за вас с помощью динамического фильтра.

13.4.1. Создание динамических фильтров

Прежде всего необходимо определить фильтр, задав его имя и параметры, которые он будет принимать во время выполнения. Это определение можно оформить в виде аннотации Hibernate перед любым классом сущности из предметной модели или в файле метаданных package-info.java:

Файл: /model/src/main/java/org/jpwh/model/filtering/dynamic/package-info.java

```
@org.hibernate.annotations.FilterDef(
    name = "limitByUserRank",
    parameters = {
        @org.hibernate.annotations.ParamDef(
            name = "currentUserRank", type = "int"
        )
    }
)
```

В этом примере фильтр называется `limitByUserRank`; обратите внимание, что имя фильтра должно быть уникальным в пределах единицы хранения. Во время выполнения он принимает один аргумент типа `int`. Если имеется несколько определений фильтров, их можно объявить в аннотации `@org.hibernate.annotations.FilterDefs`.

Пока фильтр неактивен; ничто не указывает, что он должен применяться к экземплярам `Item`. Мы должны реализовать фильтр и применить его к тем классам и коллекциям, которые нужно фильтровать.

13.4.2. Применение фильтра

Нам требуется применить заданный фильтр к классу `Item`, чтобы ни один экземпляр не был доступен пользователям, не имеющим соответствующего ранга:

Файл: /model/src/main/java/org/jpwh/model/filtering/dynamic/Item.java

```
@Entity
@org.hibernate.annotations.Filter(
    name = "limitByUserRank",
    condition =
        ":currentUserRank >= (" +
                "select u.RANK from USERS u " +
                "where u.ID = SELLER_ID" +
            ")"
)
public class Item {

    // ...
}
```

Условие `condition` представлено выражением SQL, которое выполняется на стороне базы данных, поэтому в нем можно использовать любой оператор или функ-

цию SQL. Оно должно возвращать значение `true` для записей, соответствующих условию. В данном примере для получения ранга (поля `rank`) продавца товара используется подзапрос. Имена столбцов, такие как `SELLER_ID`, относятся к таблице, отображаемой на класс сущности. Если ранг текущего пользователя окажется ниже ранга, который вернет подзапрос, экземпляр `Item` не пройдет фильтрацию. Есть возможность объединить несколько фильтров с помощью аннотации `@org.hibernate.annotations.Filters`.

Если после определения и применения фильтра активировать его в конкретной единице работы, он начнет фильтровать все экземпляры `Item`, не соответствующие условию. Давайте активируем его.

13.4.3. Активация фильтра

Мы определили фильтр данных и применили его к классу хранимой сущности. Но это еще не полноценный фильтр – он должен быть активирован и параметризован в рамках конкретной единицы работы с помощью `Session` API:

Файл: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
org.hibernate.Filter filter = em.unwrap(Session.class)
    .enableFilter("limitByUserRank");

filter.setParameter("currentUserRank", 0);
```

Активация фильтра производится по имени; метод возвращает объект `Filter`, параметры которому передаются динамически во время выполнения. Вы должны сами передать значения параметров; здесь устанавливается ранг, равный 0. В данном примере отфильтровываются все товары (`Item`), проданные пользователями (`User`), имеющие больший ранг.

В классе `Filter` определено еще несколько полезных методов, таких как `getFilterDefinition()` (позволяет последовательно обращаться к именам параметров и типам) и `validate()` (возбуждает исключение `HibernateException`, если забыть установить значение параметра). Также можно передать список аргументов с помощью метода `setParameterList()`; это очень удобно, если ограничение SQL содержит выражение с операторами, работающими на множествах (например, с оператором `IN`).

Теперь каждый запрос на основе критериев или JPQL, выполняемый в рамках фильтруемого контекста хранения, будет возвращать отфильтрованное множество экземпляров `Item`:

Файл: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM where 0 >=
//  (select u.RANK from USERS u where u.ID = SELLER_ID)
```

Файл: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();
```

```
criteria.select(criteria.from(Item.class));
List<Item> items = em.createQuery(criteria).getResultList();
// select * from ITEM where 0 >=
//  (select u.RANK from USERS u where u.ID = SELLER_ID)
```

Обратите внимание, как Hibernate динамически добавляет условия ограничений SQL в сформированные запросы.

При первом знакомстве с динамическими фильтрами вы, возможно, столкнетесь с особенностями извлечения по идентификатору. Вы, наверное, ожидаете, что результат вызова `em.find(Item.class, ITEM_ID)` также будет отфильтрован. Однако это не так: Hibernate не применяет фильтров к операциям извлечения по идентификатору. Одна из причин заключается в том, что условия фильтрации – это фрагменты SQL, а поиск по идентификатору может полностью осуществляться целиком в кэше контекста хранения первого уровня. Аналогичные рассуждения применимы и к связям *многие к одному* и *один к одному*. Если бы связь *многие к одному* подвергалась фильтрации (возвращая, например, `null` при вызове `anItem.getSeller()`), тогда изменилась бы множественность связи! Вы бы не узнали, есть ли у товара продавец, если бы не могли его увидеть.

Но вы можете динамически ограничивать доступ к коллекциям. Помните, что хранимые коллекции – это лишь псевдонимы для запросов.

13.4.4. Фильтрация коллекций

До сих пор вызов метода `someCategory.getItems()` возвращал все товары (`Item`), принадлежащие конкретной категории (`Category`). Но можно добавить ограничение, применив для коллекции фильтр:

Файл: /model/src/main/java/org/jpwh/model/filtering/dynamic/Category.java

```
@Entity
public class Category {

    @OneToMany(mappedBy = "category")
    @org.hibernate.annotations.Filter(
        name = "limitByUserRank",
        condition =
            ":currentUserRank >= (" +
                    "select u.RANK from USERS u " +
                    "where u.ID = SELLER_ID" +
                ")"
    )
    protected Set<Item> items = new HashSet<Item>();

    // ...
}
```

Если теперь активировать фильтр в рамках сеанса (`Session`), объекты, возвращаемые во время обхода коллекции `Category#items`, будут отфильтрованы:

Файл: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
filter.setParameter("currentUserRank", 0);
Category category = em.find(Category.class, CATEGORY_ID);
assertEquals(category.getItems().size(), 1);
```

Если ранг текущего пользователя равен 0, при загрузке коллекции будет получен лишь один экземпляр `Item`. Изменив значение ранга на 100, вы увидите больше данных:

Файл: /examples/src/test/java/org/jpwh/test/filtering/DynamicFilter.java

```
filter.setParameter("currentUserRank", 100);
category = em.find(Category.class, CATEGORY_ID);
assertEquals(category.getItems().size(), 2);
```

Вы, должно быть, заметили, что условие SQL одинаково в обоих случаях применения фильтра. Если ограничение SQL одинаково для всех случаев применения фильтра, его можно установить в качестве условия по умолчанию, чтобы избежать повторения:

Файл: /model/src/main/java/org/jpwh/model/filtering/dynamic/package-info.java

```
@org.hibernate.annotations.FilterDef(
    name = "limitByUserRankDefault",
    defaultCondition=
        ":currentUserRank >= (" +
                "select u.RANK from USERS u " +
                "where u.ID = SELLER_ID" +
            ")",
    parameters = {
        @org.hibernate.annotations.ParamDef(
            name = "currentUserRank", type = "int"
        )
    }
)
```

Существует множество других интересных вариантов применения динамической фильтрации данных. Вы уже видели ограничение доступа к данным на основе произвольного условия, касающегося безопасности. Это может быть ранг пользователя, принадлежность к определенной группе или назначенная пользователю роль. Данные могут храниться вместе с кодом региона (например, все деловые контакты отдела продаж). Или, к примеру, каждый менеджер по продажам мог бы работать только с данными, относящимися к его региону.

13.5. Резюме

- Каскадная передача изменений состояния – это предопределенные реакции механизма хранения на события жизненного цикла.

- Вы узнали, как принимать и обрабатывать события. Создали реализацию приемников и обработчиков событий, чтобы выполнить дополнительную логику в процессе загрузки и сохранения данных. Познакомились с методами обратного вызова JPA и механизмом расширения `Interceptor`, а также с ядром системы событий Hibernate.
- Вы можете использовать Hibernate Envers для ведения журнала аудита и хранения нескольких версий данных в базе (как в системе контроля версий). При работе с Envers копия данных автоматически сохраняется в отдельных таблицах, когда происходит добавление, изменение или удаление данных из таблиц. Envers объединяет все изменения, произведенные в транзакции, в отдельный набор изменений со своим номером версии. Вы можете использовать запросы к Envers и извлекать архивные данные.
- Используя динамическую фильтрацию данных, Hibernate может автоматически добавлять ограничения SQL к формируемым запросам.

Часть IV

СОЗДАНИЕ ЗАПРОСОВ

В части IV мы познакомимся с особенностями, а также подробно изучим API и языки запросов. Не все главы в этой части написаны как руководство; мы рассчитываем, что вы будете часто заглядывать в эту часть книги во время разработки приложений в поисках решений проблем с конкретными запросами.

Начиная с главы 14, мы будем рассказывать о создании и выполнении запросов с помощью базового API, а также об оптимизации их выполнения. В главе 15 мы обсудим различные языки запросов, покажем, как создавать запросы JPQL и на основе критериев, расскажем, как эффективно извлекать данные с помощью соединений и как создавать отчеты с применением запросов и подзапросов. В главе 16 мы углубимся в изучение более продвинутых возможностей: преобразование результатов, фильтрацию коллекций и создание запросов на основе критериев с помощью Hibernate API. Наконец, в главе 17 мы опишем такие приемы, как использование JDBC, отображение результатов запросов SQL, настройка операций создания, чтения, изменения, удаления (CRUD), а также вызов хранимых процедур.

После прочтения этой части книги вы сможете извлекать любую информацию из базы данных, используя различные варианты запросов, настраивая доступ к данным по необходимости.

Глава 14

Создание и выполнение запросов

В этой главе:
- базовый API запросов;
- создание и подготовка запросов;
- оптимизация выполнения запросов.

Если вы много лет писали SQL-запросы вручную, вас наверняка беспокоит возможность лишиться привычной гибкости и выразительности. Но, смеем вас уверить, при использовании Hibernate и Java Persistence этого не произойдет.

Мощные механизмы запросов в Hibernate и Java Persistence позволяют выразить практически все, что вам часто (или нечасто) приходилось делать с помощью SQL, но используя при этом объектно-ориентированный подход – с помощью классов и полей. Более того, вы всегда сможете вернуться к SQL и переложить на Hibernate всю тяжелую работу по обработке результатов запроса. Дополнительные ресурсы, посвященные SQL, находятся в справочном разделе.

Главные нововведения в JPA 2

- Прикладной интерфейс для программного создания типизированных запросов на основе критериев.
- С помощью интерфейса `TypedQuery` теперь можно объявить тип результата запроса в самом начале.
- Можно программно сохранить объект `Query` (JPQL, запрос на основе критериев или обычный запрос SQL) для повторного использования в качестве именованного запроса.
- В дополнение к возможности задавать параметры, подсказки, максимальное количество результатов, режим выталкивания контекста и блокировки JPA 2 добавляет в `Query` API разнообразные методы чтения для получения текущих настроек.
- В JPA стандартизировано несколько рекомендаций для запросов (максимальное время ожидания, использование кэширования).

В этой главе мы покажем, как создавать и выполнять запросы с помощью JPA и Hibernate API. Запросы будут максимально простыми, чтобы вы могли сосредоточиться на API для их создания и выполнения, не отвлекаясь на незнакомые языки запросов (они будут рассмотрены в следующей главе).

Общим для всех API является необходимость подготовки запроса в коде приложения перед выполнением, включающей три шага:

1) создание запроса с произвольной выборкой, ограничениями и проекциями извлекаемых данных;
2) подготовка запроса: связывание аргументов с параметрами запроса, установка подсказок и настройка постраничного вывода. Вы можете повторно использовать запрос, изменяя его настройки;
3) выполнение подготовленного запроса в базе данных и извлечение информации. Вы можете управлять выполнением запроса и способом загрузки данных в память (например, загружать все сразу или по частям).

В зависимости от разновидности отправной точкой для создания запроса будет `EntityManager` или `Session`. Первый шаг – создание запроса.

14.1. Создание запросов

JPA представляет запрос в виде объекта `javax.persistence.Query` или `javax.persistence.TypedQuery`. Запросы создаются с помощью метода `EntityManager#createQuery()` или его вариантов. Запрос может быть написан на языке Java Persistence Query Language (JPQL), создан при помощи объекта `CriteriaBuilder` или интерфейса `CriteriaQuery` API или написан на обычном SQL (также доступен интерфейс `javax.persistence.StoredProcedureQuery`, который рассматривается в разделе 17.4).

В Hibernate есть свои, более старые API запросов: `org.hibernate.Query` и `org.hibernate.SQLQuery`. Мы поговорим о них чуть ниже, но сначала рассмотрим стандартные интерфейсы и языки запросов JPA.

14.1.1. Интерфейсы запросов JPA

Предположим, что требуется извлечь все экземпляры `Item` из базы данных. Этот простой запрос на языке JPQL выглядит очень похожим на SQL:

```
Query query = em.createQuery("select i from Item i");
```

Реализация JPA возвращает новый объект `Query`, но Hibernate еще не отправил код SQL в базу данных. Помните, что дальнейшая подготовка и выполнение запроса являются отдельными шагами.

Язык JPQL прост и понятен каждому, имеющему опыт использования SQL. Вместо имен столбцов и таблиц язык JPQL оперирует именами классов и свойств сущностей. Не считая имен классов и свойств, JPQL не чувствителен к регистру, поэтому не важно, напишите вы `SeLECt` или `select`.

Строки запросов JPQL (и SQL) могут быть простыми литералами в коде, как в предыдущем примере. С другой стороны, особенно в больших приложениях, строки запросов можно убрать из кода доступа к данным и поместить их в аннота-

ции или в файлы XML. Потом обратиться к запросу можно вызовом `EntityManager#createNamedQuery()`. Мы отдельно обсудим тему хранения запросов в отдельных файлах далее в этой главе; существует много нюансов, которые нужно учитывать.

Наибольшее неудобство при работе с JPQL доставляет рефакторинг предметной модели: если переименовать класс `Item`, ваш запрос JPQL перестанет работать (хотя некоторые IDE способны обнаруживать строки JPQL и проводить их рефакторинг).

JPA и языки запросов: HQL против JPQL

До появления JPA (а в некоторой документации и по сей день) язык запросов в Hibernate назывался HQL. На сегодняшний день различия между JPQL и HQL сгладились. Каждый раз, передавая строку запроса любому интерфейсу запросов Hibernate, будь то `EntityManager` или `Session`, вы передаете строку JPQL/HQL. Внутри эти строки обрабатывает один и тот же парсер. Основной синтаксис и семантика совпадают, хотя, как обычно, Hibernate поддерживает некоторые конструкции, не стандартизованные в JPA. Мы сообщим вам, если какое-то ключевое слово или предложение в примере будет работать только с Hibernate. Для простоты, увидев аббревиатуру *HQL*, мысленно подставляйте вместо нее *JPQL*.

С помощью интерфейсов `CriteriaBuilder` и `CriteriaQuery` можно создавать полностью типизированные запосы. В JPA эти запросы также называются *запросами на основе критериев*:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
// Можно использовать EntityManagerFactory:
// CriteriaBuilder cb = entityManagerFactory.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery();
criteria.select(criteria.from(Item.class));

Query query = em.createQuery(criteria);
```

Сначала нужно получить экземпляр `CriteriaBuilder` из объекта `EntityManager`, вызвав `getCriteriaBuilder()`. Если у вас нет готового экземпляра `EntityManager` (например, когда требуется создать запрос отдельно от конкретного контекста хранения), экземпляр `CriteriaBuilder` можно получить из объекта `EntityManagerFactory`, обычно доступный глобально.

Затем можно использовать экземпляр построителя (builder) и создать любое количество экземпляров `CriteriaQuery`. Для каждого объекта `CriteriaQuery` должен быть указан хотя бы один корневой класс с помощью метода `from()`, как `Item.class` в предыдущем примере. Это называется *выборкой*; мы расскажем больше о выборках в следующей главе. Запрос в примере возвращает все экземпляры `Item` из базы данных.

Интерфейс `CriteriaQuery` можно без проблем встроить в приложение, не используя операций со строками. Это лучший вариант, когда невозможно до конца определить запрос во время разработки и приложение должно формировать

его динамически во время выполнения. Представьте, что требуется реализовать в приложении поиск по маске, используя различные флажки, поля ввода и переключатели, которые может менять пользователь. Запрос к базе данных должен быть сформирован автоматически, на основе указанных пользователем параметров поиска. При использовании JPQL и конкатенации строк такой код трудно будет написать и поддерживать.

Вы можете создать строго типизированный объект `CriteriaQuery` безо всяких строк, используя статическую метамодель JPA. Это означает, что ваши запросы останутся в безопасности и смогут подвергаться рефакторингу, как уже было показано в разделе «Статическая метамодель» в главе 3.

Создание отсоединенного запроса на основе критериев

Чтобы получить экземпляр JPA `CriteriaBuilder`, необходим объект `EntityManager` или `EntityManagerFactory`. При работе с более старым оригинальным интерфейсом `org.hibernate.Criteria`, чтобы создать отсоединенный запрос, необходим лишь доступ к корневому классу сущности, как показано в разделе 16.3.

Когда необходимо использовать возможности, доступные только для вашей базы данных, единственный вариант – использовать обычный SQL. Вы можете непосредственно выполнить код SQL и возложить на Hibernate обработку результата с помощью метода `EntityManager#createNativeQuery()`:

```
Query query = em.createNativeQuery(
    "select * from ITEM", Item.class
);
```

После выполнения этого запроса SQL Hibernate на основе данных в объекте `java.sql.ResultSet` создаст коллекцию `List` управляемых экземпляров `Item`. Все столбцы, необходимые для создания объекта `Item`, конечно же, должны присутствовать в результате; если запрос SQL не вернет их, будет возбуждено исключение.

На практике большинство запросов в вашем приложении будет довольно простым – вы сможете легко написать их с помощью JPQL или `CriteriaQuery`. Затем, возможно во время оптимизации, вы обнаружите несколько сложных запросов, для которых очень важна производительность. Возможно, вам потребуется использовать специальные ключевые слова SQL для управления оптимизатором вашей СУБД. В таких случаях большинство разработчиков пишет запросы на чистом SQL вместо JPQL, сохраняя такие сложные запросы в файлах XML, где их можно менять независимо от Java-кода с помощью администратора баз данных. Hibernate по-прежнему сможет обрабатывать результаты запроса за вас. Нет ничего плохого в том, чтобы использовать SQL в приложении Hibernate; не дайте ORM-«пуризму» помешать вам в этом. Сталкиваясь с особым случаем, не пытайтесь его спрятать, лучше выделите его особо и хорошо задокументируйте, чтобы следующий инженер понимал, что происходит.

Порой бывает полезно указывать тип данных, возвращаемых запросом.

14.1.2. Результаты типизированных запросов

Предположим, нужно извлечь только один экземпляр `Item` по заданному идентификатору:

```
Query query = em.createQuery(
    "select i from Item i where i.id = :id"
).setParameter("id", ITEM_ID);

Item result = (Item) query.getSingleResult();
```

Этот пример демонстрирует порядок связывания параметров и выполнение запроса. Обратите внимание на значение, возвращаемое методом `getSingleResult()`. Оно имеет тип `java.lang.Object`, и поэтому требуется выполнить приведение к типу `Item`.

Но если во время создания запроса указать класс возвращаемого значения, приведение типа будет не нужно. Эту работу выполнит интерфейс `javax.persistence.TypedQuery`:

```
TypedQuery<Item> query = em.createQuery(
    "select i from Item i where i.id = :id", Item.class
).setParameter("id", ITEM_ID);

Item result = query.getSingleResult();  ← Приведение типа не нужно
```

Интерфейс `TypedQuery` также поддерживается запросами на основе критериев:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(cb.equal(i.get("id"), ITEM_ID));

TypedQuery<Item> query = em.createQuery(criteria);

Item result = query.getSingleResult();  ← Приведение типа не нужно
```

Обратите внимание, что этот запрос `CriteriaQuery` не является полностью типизированным: имя свойства `Item#id` задается с помощью строки в методе `get("id")`. В главе 3, в разделе «Статическая метамодель», вы видели, как создавать полностью типобезопасные запросы с помощью статической метамодели классов.

Hibernate старше самой первой версии JPA, поэтому в нем есть свои API для работы с запросами.

Особенности Hibernate

14.1.3. Интерфейсы Hibernate для работы с запросами

В Hibernate запросы представлены объектами `org.hibernate.Query` и `org.hibernate.SQLQuery`. Как обычно, они имеют больше возможностей, чем определено в JPA, но за это приходится расплачиваться переносимостью. Эти интерфейсы гораздо старше, чем JPA, поэтому имеет место повторение функциональности.

Чтобы воспользоваться Hibernate API для работы с запросами, требуется сначала получить экземпляр `Session`:

```
Session session = em.unwrap(Session.class);
org.hibernate.Query query = session.createQuery("select i from Item i");
// Нестандартный API: query.setResultTransformer(...);
```

Запрос можно написать на обычном JPQL. Сравнивая `javax.persistence.Query` с `org.hibernate.Query`, можно обнаружить несколько дополнительных нестандартных методов, доступных только в Hibernate. Вы узнаете больше об этом API далее в этой и следующих главах.

В Hibernate также имеется собственный механизм отображения результатов запросов SQL, доступный в интерфейсе `org.hibernate.SQLQuery`:

```
Session session = em.unwrap(Session.class);

org.hibernate.SQLQuery query = session.createSQLQuery(
    "select {i.*} from ITEM {i}"
).addEntity("i", Item.class);
```

Этот пример работает за счет использования символов подстановки в тексте запроса SQL для отображения столбцов `java.sql.ResultSet` в свойства сущности. В разделе 17.2 мы еще затронем тему интеграции запросов SQL с данным нестандартным механизмом и с механизмом отображения JPA.

В Hibernate также имеется более старый интерфейс для создания запросов на основе критериев – `org.hibernate.Criteria`:

```
Session session = em.unwrap(Session.class);

org.hibernate.Criteria query = session.createCriteria(Item.class);
query.add(org.hibernate.criterion.Restrictions.eq("id", ITEM_ID));

Item result = (Item) query.uniqueResult();
```

Чтобы получить доступ к Hibernate API для работы с запросами из интерфейса `javax.persistence.Query`, нужно сначала преобразовать его в `org.hibernate.jpa.HibernateQuery`:

```
javax.persistence.Query query = em.createQuery(
    // ...
);

org.hibernate.Query hibernateQuery =
    query.unwrap(org.hibernate.jpa.HibernateQuery.class)
        .getHibernateQuery();

hibernateQuery.getQueryString();
hibernateQuery.getReturnAliases();
// ... прочие вызовы проприетарного API
```

Далее мы сначала рассмотрим стандартный API, а затем покажем некоторые продвинутые и редко используемые возможности, доступные только в Hibernate, такие как *прокрутка с помощью курсоров* или *запросы на основе примеров*.

После создания запроса, но перед выполнением вам, возможно, понадобится установить его параметры.

14.2. Подготовка запросов

Запрос одновременно решает несколько задач: определяет информацию, которая должна быть загружена из базы данных, а также накладывает на нее ограничения, которым она должна соответствовать, такие как идентификатор товара (`Item`) или имя пользователя (`User`). При создании запроса необязательно внедрять эти параметры в текст запроса, используя конкатенацию строк. Вместо этого предпочтительнее использовать параметры и связать с ними значения аргументов перед выполнением. Это позволяет повторно использовать запрос с разными значениями аргументов, исключая возможность атак, основанных на внедрении SQL-кода.

В зависимости от требований пользовательского интерфейса может потребоваться реализовать *постраничную выборку*. Для этого можно ограничить количество записей, возвращаемых запросом. Например, можно вернуть записи с 1 по 20 – ровно столько, сколько можно показать на экране, – а позже извлечь строки с 21 по 40 и т. д.

Начнем со связывания параметров.

14.2.1. Защита от атак на основе внедрения SQL-кода

Не имея возможности связывать параметры во время выполнения, вы вынуждены писать плохой код:

```
String searchString = getValueEnteredByUser();  ← Никогда не делайте так!

Query query = em.createQuery(
    "select i from Item i where i.name = '" + searchString + "'"
);
```

Никогда не пишите подобного кода, поскольку злоумышленник может сконструировать строку поиска, которая позволит ему выполнить произвольный код в базе данных, например введя значение переменной `searchString` в поле поиска как `foo' and callSomeStoredProcedure() and 'bar' = 'bar`.

Как видите, `searchString` – уже не просто строка с поисковым запросом, а код, вызывающий хранимую процедуру в базе данных! Символы кавычек не экранируются; следовательно, вызов хранимой процедуры – это еще одно корректное выражение в запросе. Конструируя текст запроса подобным образом, вы создаете огромную брешь в безопасности приложения, позволяя выполнять произвольный код на стороне базы данных. Это и есть атака на основе *внедрения кода SQL*. Никогда не передавайте непроверенных данных от пользователя в базу данных! К счастью, есть простой механизм, предотвращающий такую ошибку.

Механизм JDBC предоставляет возможность безопасного связывания значений с параметрами SQL. Он точно знает, какие символы в значениях аргументов нужно экранировать, благодаря чему предыдущая уязвимость становится просто

невозможной. К примеру, драйвер базы данных экранирует символы одинарных кавычек в строке `searchString` и обращается с ними не как с управляющими символами, а как с частью строки поиска. Кроме того, при использовании параметров база данных может эффективно кэшировать предварительно скомпилированные выражения, значительно увеличивая производительность.

Существуют два подхода к связыванию параметров: по *именам* и по *позициям*. JPA поддерживает оба варианта, но их нельзя использовать одновременно в одном запросе.

14.2.2. Связывание именованных параметров

Используя именованные параметры, предыдущий запрос можно переписать так:

```
String searchString = // ...

Query query = em.createQuery(
    "select i from Item i where i.name = :itemName"
).setParameter("itemName", searchString);
```

Двоеточие, за которым следует имя, определяет именованный параметр `itemName`. Следующий шаг – связать значение с параметром `itemName`. Такой код выглядит опрятнее, он безопаснее и обладает лучшей производительностью, поскольку вы можете повторно использовать скомпилированное выражение SQL, меняя лишь значение параметра.

Вы можете получить множество (`Set`) всех объектов `Parameter` из экземпляра запроса (`Query`), чтобы узнать больше о каждом параметре (например, имя или Java-тип) или убедиться перед выполнением запроса, что все параметры связаны со значениями:

```
for (Parameter<?> parameter : query.getParameters()) {
    assertTrue(query.isBound(parameter));
}
```

Метод `setParameter()` – это обобщенная операция, которая может связывать аргменты всех типов. Ей нужно только немного помочь при использовании значений времени:

```
Date tomorrowDate = // ...

Query query = em.createQuery(
    "select i from Item i where i.auctionEnd > :endDate"
).setParameter("endDate", tomorrowDate, TemporalType.TIMESTAMP);
```

Для Hibernate важно знать, передавать ли дату, время или отметку времени.

Также ради удобства можно передать в метод `setParameter()` экземпляр:

```
Item someItem = // ...

Query query = em.createQuery(
    "select b from Bid b where b.item = :item"
).setParameter("item", someItem);
```

Hibernate свяжет значение идентификатора переданного экземпляра `Item`. Далее вы увидите, что выражение `b.item` – это лишь сокращенная форма записи `b.item.id`.

Для запросов на основе критериев существуют два способа связывания параметров, простой и сложный:

```
String searchString = // ...

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);

Query query = em.createQuery(
    criteria.select(i).where(
        cb.equal(
            i.get("name"),
            cb.parameter(String.class, "itemName")
        )
    )
).setParameter("itemName", searchString);
```

Здесь мы добавили в экземпляр `CriteriaQuery` параметр `itemName` типа `String`, а затем связали его со значением, используя обычный метод `Query#setParameter()`.

Можно также применить объект `ParameterExpression`, избавляющий от необходимости придумывать имя для параметра и обеспечивающий безопасность типов (нельзя связать значение типа `Integer` с объектом `ParameterExpression<String>`):

```
String searchString = // ...

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery(Item.class);
Root<Item> i = criteria.from(Item.class);

ParameterExpression<String> itemNameParameter =
    cb.parameter(String.class);

Query query = em.createQuery(
    criteria.select(i).where(
        cb.equal(
            i.get("name"),
            itemNameParameter
        )
    )
).setParameter(itemNameParameter, searchString);
```

Менее безопасный и редко используемый способ связывания значений – с использованием позиционных параметров.

14.2.3. Связывание позиционных параметров

При желании вместо именованных параметров можно использовать позиционные:

```
Query query = em.createQuery(
    "select i from Item i where i.name like ?1 and i.auctionEnd > ?2"
);
query.setParameter(1, searchString);
query.setParameter(2, tomorrowDate, TemporalType.TIMESTAMP);
```

В этом примере позиционные параметры отмечены индексами `?1` и `?2`. Похожие обозначения параметров используются в JDBC, но только с одними знаками вопросов без номеров. JPA требует нумерации параметров, начиная с 1.

> **ПРИМЕЧАНИЕ** В Hibernate поддерживаются оба подхода, поэтому будьте внимательны! Hibernate предупредит о ненадежном запросе, если вы будете использовать позиционные параметры в стиле JDBC, обозначаемые только знаками вопросов.

Мы рекомендуем избегать позиционных параметров. Возможно, их удобно использовать для построения сложных запросов, но для этой цели лучше подходит `CriteriaQuery` API.

После связывания параметров запроса можно настроить *постраничную выборку*, если невозможно сразу отобразить всех результатов.

14.2.4. Постраничная выборка больших наборов с результатами

Для обработки больших наборов с результатами запросов на практике широко используется *постраничная выборка*. Пользователи могут просматривать результаты своих поисковых запросов (например, конкретные товары) постранично. На каждой странице отображается ограниченное подмножество (например, 10 товаров), и пользователи могут самостоятельно переходить к следующим и предыдущим страницам для просмотра остальных результатов запроса.

Интерфейс `Query` поддерживает постраничную выборку. В следующем примере требуемая страница начинается в середине множества с результатами:

```
Query query = em.createQuery("select i from Item i");
query.setFirstResult(40).setMaxResults(10);
```

Этот код извлечет следующие десять записей, начиная с сороковой. Вызов `setFirstResults(40)` указывает, что страница начинается с записи № 40 во множестве результатов. Вызов `setMaxResults(10)` ограничивает количество записей, возвращаемых базой данных. Поскольку в SQL отсутствует стандартный способ представления постраничной выборки, Hibernate использует особые приемы для эффективной работы с конкретными СУБД.

Чрезвычайно важно помнить, что постраничная выборка реализована на уровне SQL для строк результата. Ограничение результата десятью строками не обязательно означает то же самое, что ограничение результата десятью экземплярами `Item`! В разделе 15.4.5 вы увидите несколько запросов с *динамическим извлечением*, которые не могут работать совместно с постраничной выборкой строк на уровне SQL, и мы снова обсудим этот вопрос.

Постраничная выборка поддерживается даже для запросов SQL:

```
Query query = em.createNativeQuery("select * from ITEM");
query.setFirstResult(40).setMaxResults(10);
```

Hibernate автоматически добавит в SQL-запрос необходимые предложения и ключевые слова, ограничивающие количество возвращаемых строк для конкретной страницы.

На практике часто приходится совмещать постраничную выборку со специальным запросом подсчета записей. Отображая страницу со списком товаров, можно также сообщить пользователю их полное количество. Кроме того, эта информация понадобится, чтобы решить, остались ли еще непросмотренные страницы, и дать пользователю возможность перейти к следующей. Обычно для этого требуется выполнить два разных запроса: например, `select i from Item i` в сочетании с методами `setMaxResults()` и `setFirstResult()` извлечет порцию результатов, а `select count(i) from Item i` вернет количество доступных товаров.

Особенности Hibernate

Старайтесь избегать дополнительных затрат на поддержку практически одинаковых запросов. Часто с этой целью пишут один запрос, но выполняют его с помощью курсора базы данных, чтобы получить общее количество результатов:

```
Query query = em.createQuery("select i from Item i");

org.hibernate.Query hibernateQuery =          ← ❶ Преобразовать API
    query.unwrap(org.hibernate.jpa.HibernateQuery.class).getHibernateQuery();

org.hibernate.ScrollableResults cursor =  ← ❷ Выполнить запрос с помощью курсора
        hibernateQuery.scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

cursor.last();    ← ❸ Подсчитать записи
int count = cursor.getRowNumber()+1;

cursor.close();   ← ❹ Закрыть курсор

query.setFirstResult(40).setMaxResults(10);   ← ❺ Получить произвольный набор данных
```

❶ Преобразование Hibernate API для использования курсора с поддержкой прокрутки.
❷ Выполнение запроса с помощью курсора базы данных; результат запроса не загружается в память.
❸ Переход к последней записи в наборе результатов в базе данных, с последующим получением количества строк. Поскольку нумерация строк начинается с нуля, чтобы получить их количество, нужно к номеру последней строки прибавить 1.
❹ Закрыть курсор в базе данных.
❺ Вновь выполнить запрос и извлечь произвольную страницу с результатами.

Но этот удобный подход имеет один существенный недостаток: ваш JDBC-драйвер и/или СУБД могут не поддерживать курсоров. Хуже того, может показаться, что курсор работает, но данные будут незаметно загружаться в память приложения; курсор работает не напрямую с базой данных. Драйверы Oracle и MySQL

известны своими проблемами, и мы еще многое скажем о прокрутке результатов запросов и курсорах в следующем разделе. Позже в этой книге, в разделе 19.2, мы еще раз обсудим стратегии постраничной выборки на стороне приложения.

Теперь запрос готов к выполнению.

14.3. Выполнение запросов

После создания и подготовки экземпляра `Query` можно выполнить запрос и загрузить результаты в память. На практике часто используют прием извлечения полного набора результатов запроса в память за одно обращение; мы называем это *выводом списка* (*listing*). Существуют также другие варианты, которые мы обсудим ниже, такие как *прокрутка с помощью курсоров* (*scrolling*) или *итераторов* (*iterating*).

14.3.1. Извлечение полного списка результатов

Метод `getResultList()` выполнит запрос (`Query`) и вернет результаты в виде коллекции `java.util.List`:

```
Query query = em.createQuery("select i from Item i");
List<Item> items = query.getResultList();
```

Hibernate сразу же выполнит одно или несколько SQL-выражений `SELECT`, в зависимости от стратегии извлечения. Если вы отобразили некоторые связи или коллекции с параметром `FetchType.EAGER`, Hibernate должен извлечь их вместе с основными данными. Все данные загружаются в память, и все экземпляры сущностей, извлекаемые Hibernate, будут находиться в хранимом состоянии под управлением контекста хранения.

Конечно, контекст хранения не управляет результатами скалярных проекций. Следующий запрос вернет список (`List`) строк (`String`):

```
Query query = em.createQuery("select i.name from Item i");
List<String> itemNames = query.getResultList();
```

Некоторые запросы возвращают только единичный результат, например когда нужна лишь самая большая ставка (`Bid`) за товар (`Item`).

14.3.2. Получение единичных результатов

Выполнить запрос, возвращающий единственный результат, можно с помощью метода `getSingleResult()`:

```
TypedQuery<Item> query = em.createQuery(
    "select i from Item i where i.id = :id", Item.class
).setParameter("id", ITEM_ID);

Item item = query.getSingleResult();
```

Вызов `getSingleResult()` вернет экземпляр `Item`. Тот же прием можно использовать для извлечения скаляров:

```
TypedQuery<String> query = em.createQuery(
    "select i.name from Item i where i.id = :id", String.class
).setParameter("id", ITEM_ID);

String itemName = query.getSingleResult();
```

А теперь неприятные подробности: в отсутствие результатов метод `getSingleResult()` возбудит исключение `NoResultException`. Следующий запрос пытается отыскать товар с несуществующим идентификатором:

```
try {
    TypedQuery<Item> query = em.createQuery(
        "select i from Item i where i.id = :id", Item.class
    ).setParameter("id", 1234l);

    Item item = query.getSingleResult();
    // ...

} catch (NoResultException ex) {
    // ...
}
```

Возможно, вы ожидали получить `null` от этого безобидного запроса. Ситуация тем неприятнее, что приходится обертывать этот участок кода блоком `try/catch`. Фактически это вынуждает *всегда* обертывать вызовы `getSingleResult()`, поскольку заранее неизвестно, сколько записей окажется в результате.

Если в результате база данных вернет несколько записей, `getSingleResult()` возбудит исключение `NonUniqueResultException`. Такое обычно происходит со следующим видом запросов:

```
try {
    Query query = em.createQuery(
        "select i from Item i where name like '%a%'"
    );

    Item item = (Item) query.getSingleResult();
    // ...

} catch (NonUniqueResultException ex) {
    // ...
}
```

Извлечение всех результатов в память на практике используется чаще всего. Но Hibernate поддерживает другие способы, которые могут вас заинтересовать, если потребуется оптимизировать потребление памяти или особенности выполнения запроса.

Особенности Hibernate

14.3.3. Прокрутка с помощью курсоров базы данных

Обычный JDBC поддерживает *наборы результатов с возможностью прокрутки*. Для этого используется курсор, открытый на стороне СУБД. Курсор указывает на конкретную запись в результатах запроса, а приложение может перемещать его вперед и назад. С помощью курсора можно даже перейти к конкретной записи.

Одна из причин, по которой может понадобиться перемещаться по результатам запроса вместо загрузки их в память, – большое количество результатов, не умещающееся в памяти. Обычно возвращаемый набор пытаются уменьшить путем наложения дополнительных ограничений в запросе. Иногда это невозможно, например если требуется вывести все данные, но в несколько заходов. Мы покажем такую процедуру пакетной обработки в разделе 20.1.

JPA не определяет порядка прокрутки результатов запроса с помощью курсоров базы данных, поэтому вам понадобится интерфейс `org.hibernate.ScrollableResults`, получаемый из нестандартного интерфейса `org.hibernate.Query`:

```
Session session = em.unwrap(Session.class);

org.hibernate.Query query = session.createQuery(   ← ❶ Открыть курсор
    "select i from Item i order by i.id asc"
);

org.hibernate.ScrollableResults cursor =   ← ❷ Создать запрос
        query.scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

cursor.setRowNumber(2);                    ← ❸ Перейти к третьей записи

Item item = (Item) cursor.get(0);          ← ❹ Получить значение столбца

cursor.close();                            ← ❺ Закрыть курсор
```

Сначала нужно создать объект `org.hibernate.Query` ❶ и открыть курсор ❷. Затем пропустить две первые записи в результатах, перейдя сразу к третьей ❸, и получить значение первого «столбца» из этой записи ❹. В JPQL нет столбцов, поэтому здесь это соответствует первому элементу проекции: `i` в предложении `select`. Другие примеры проекций вы найдете в следующей главе. *Всегда* закрывайте курсор ❺ до завершения транзакции в базе данных!

Как упоминалось ранее в этой главе, из обычного интерфейса `javax.persistence.Query`, созданного при помощи `CriteriaBuilder`, с помощью метода `unwrap()` можно получить Hibernate API для запросов. Запрос на основе интерфейса `org.hibernate.Criteria` также поддерживает прокрутку с помощью курсоров; возвращаемый курсор типа `ScrollableResults` работает аналогично.

Константы, определяемые перечислением `ScrollMode` из Hibernate API, соответствуют константам JDBC. Значение `ScrollMode.SCROLL_INSENSITIVE` в предыдущем

примере означает, что никакие изменения в базе данных не повлияют на курсор, т. е. результат запроса не подвержен неповторимому или фантомному чтению. Два оставшихся режима – `SCROLL_SENSITIVE` (курсор, чувствительный к изменениям) и `FORWARD_ONLY` (курсор, допускающий перемещение только вперед). Курсор, чувствительный к изменениям, отображает подтвержденные изменения в базе данных, пока он открыт; курсор, допускающий перемещение только вперед, не может ссылаться на абсолютную позицию в результатах запроса. Обратите внимание, что кэш контекста хранения в Hibernate по-прежнему обеспечивает повторимость чтения экземпляров сущности даже при использовании курсора, чувствительного к изменениям, поэтому использование такого курсора позволит видеть только изменившиеся скалярные значения, выбираемые с помощью проекций.

Имейте в виду, что не все драйверы JDBC поддерживают прокрутку с помощью курсоров, хотя может казаться, что они работают. К примеру, драйверы MySQL сразу загружают полный набор результатов в память; поэтому прокрутка происходит в памяти приложения. Чтобы по-настоящему извлекать запись за записью, нужно установить размер извлекаемой выборки JDBC в значение `Integer.MIN_VALUE` (как объяснялось в разделе 14.5.4) и задать режим `ScrollMode.FORWARD_ONLY`.

Исследуйте поведение и документацию для вашей СУБД и драйвера JDBC перед использованием курсоров.

Важным ограничением прокрутки с помощью курсоров в базе данных является невозможность объединения с динамическим извлечением, задаваемым предложением `join fetch` в JPQL. Связанное извлечение работает с несколькими записями одновременно, поэтому вы не сможете извлекать данные запись за записью. Hibernate возбудит исключение при вызове `scroll()` для запроса, использующего динамическое извлечение.

Другой способ извлечения данных требует применения итератора (*iteration*).

Особенности Hibernate

14.3.4. Обход результатов с применением итератора

Предположим, что вам известно, что большинство экземпляров сущностей, извлекаемых запросом, уже находится в памяти. Они могут находиться в контексте хранения или в общем кэше второго уровня (см. раздел 20.2). В таком случае может иметь смысл *последовательный обход* результатов запроса с помощью нестандартного `org.hibernate.Query` API:

```
Session session = em.unwrap(Session.class);

org.hibernate.Query query = session.createQuery(
    "select i from Item i"
);

Iterator<Item> it = query.iterate(); // select ID from ITEM
while (it.hasNext()) {
```

```
    Item next = it.next(); // select * from ITEM where ID = ?
    // ...
}

Hibernate.close(it);  ← Итератор должен быть закрыт вместе с экземпляром Session или вручную
```

При вызове `query.iterate()` Hibernate выполнит запрос, отправив SQL-выражение `SELECT` в базу данных. Но Hibernate слегка модифицирует запрос, и вместо всех столбцов из таблицы `ITEM` получит только значения идентификаторов/первичных ключей.

Затем, при каждом вызове `next()` интерфейса `Iterator`, будет выполняться дополнительный запрос SQL и загружаться оставшаяся часть записи из таблицы `ITEM`. Очевидно, что это приведет к проблеме *n + 1 выражений SELECT*, если только Hibernate не удастся избежать дополнительных запросов при вызове `next()`. А это произойдет, только если Hibernate найдет данные экземпляра в кэше контекста хранения или в кэше второго уровня.

Объект `Iterator`, возвращаемый методом `iterate()`, должен быть закрыт. Hibernate закроет его автоматически, при закрытии `EntityManager` или `Session`. Если есть риск в процедуре, использующей итераторы, превысить максимальное количество открытых курсоров в базе данных, объект `Iterator` можно закрыть вручную, вызвав метод `Hibernate.close(iterator)`.

Применение итераторов редко бывает оправдано, учитывая, что для эффективной работы процедуры из примера все аукционные товары должны находиться в кэше. Как и в случае с курсором, вы не сможете использовать динамического извлечения и предложения `join fetch`, потому что в таком случае Hibernate возбудит исключение.

До сих пор в каждом примере мы использовали строковые литералы Java с текстом запроса. Это подходит только для небольших запросов, но когда в работу включаются сложные, многострочные запросы, это становится неудобным. Вместо этого можно дать каждому запросу имя и перенести его либо в аннотации, либо в файлы XML.

14.4. Обращение к запросам по именам и их удаление из программного кода

Удаление запросов из программного кода позволяет хранить все запросы, относящиеся к конкретному хранимому классу (или набору классов), вместе с остальными метаданными класса. Другой подход заключается в перемещении всех запросов в файл XML. Такой подход предпочтительнее в больших приложениях; сотни запросов легче поддерживать, когда они расположены в нескольких хорошо известных местах, а не разбросаны по всему коду различных классов, обращающихся к базе данных. Обращение к таким запросам происходит по именам.

14.4.1. Вызов именованных запросов

Метод `EntityManager#getNamedQuery()` возвращает объект именованного запроса `Query`:

```
Query query = em.createNamedQuery("findItems");
```

Также можно получить экземпляр типизированного именованного запроса `TypedQuery`:

```
TypedQuery<Item> query = em.createNamedQuery("findItemById", Item.class);
```

Интерфейс запросов Hibernate также позволяет получать именованные запросы:

```
org.hibernate.Query query = session.getNamedQuery("findItems");
```

Именованные запросы являются глобальными, т. е. имя запроса является его уникальным идентификатором в пределах единицы хранения или экземпляра `org.hibernate.SessionFactory`. Коду приложения совершенно не важно, определены они в файлах XML или в аннотациях. Во время запуска Hibernate загрузит именованные запросы JPQL из файлов XML и/или из аннотаций и проверит их синтаксическую корректность. (Это может быть полезно на этапе разработки, но в рабочей версии для уменьшения времени загрузки можно избавиться от этого поведения, добавив в настройки единицы хранения параметр `hibernate.query.startup_check`.)

Не важно, на каком языке создается именованный запрос. Это может быть и JPQL, и SQL.

14.4.2. Хранение запросов в метаданных XML

Именованный запрос можно поместить в любой JPA-элемент `<entity-mappings>` в файле метаданных orm.xml. В более крупных приложениях мы советуем хранить именованные запросы отдельно, каждый в своем файле. Также можно использовать один и тот же файл отображения XML для определения запросов и конкретных классов.

Элемент `<named-query>` определяет именованный запрос JPQL:

Файл: /model/src/main/resources/querying/ExternalizedQueries.xml

```
<entity-mappings
   version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
           http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

    <named-query name="findItems">
        <query><![CDATA[
            select i from Item i
        ]]></query>
```

```
    </named-query>
</entity-mappings>
```

Текст запроса нужно обернуть инструкциями `CDATA`, чтобы ни один из его символов, который может быть случайно воспринят как XML (например, оператор *меньше*), не помешал работе синтаксического анализатора XML. Для краткости мы не будет использовать `CDATA` в большинстве примеров.

Именованные запросы не обязательно создавать с помощью JPQL. Это могут быть обычные запросы SQL; программному коду на Java эти различия не интересны:

Файл: /model/src/main/resources/querying/ExternalizedQueries.xml

```
<named-native-query name="findItemsSQL"
                    result-class="org.jpwh.model.querying.Item">
    <query>select * from ITEM</query>
</named-native-query>
```

Это полезно, если позднее потребуется оптимизировать запросы с помощью тонкой настройки SQL. Это может также пригодиться при модификации унаследованного приложения для работы с JPA/Hibernate, где код SQL может быть изолирован от написанных вручную процедур JDBC. Используя именованные запросы, их можно легко переносить один за другим в файлы отображений.

Особенности Hibernate

В Hibernate есть собственное нестандартное средство для хранения запросов в XML-файлах метаданных Hibernate:

Файл: /model/src/main/resources/querying/ExternalizedQueries.hbm.xml

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <query name="findItemsOrderByAuctionEndHibernate">
        select i from Item i order by i.auctionEnd asc
    </query>

    <sql-query name="findItemsSQLHibernate">
        <return class="org.jpwh.model.querying.Item"/>
        select * from ITEM order by NAME asc
    </sql-query>

</hibernate-mapping>
```

Вы еще увидите примеры перемещения различных запросов SQL в отдельные файлы в главе 17.

Если вам не по душе XML, запросы можно определять с помощью Java-аннотаций.

14.4.3. Хранение запросов в аннотациях

JPA поддерживает именованные запросы с помощью аннотаций `@NamedQuery` и `@NamedNativeQuery`. Но эти аннотации должны находиться перед отображаемым классом. Обратите внимание, что имя запроса также должно быть уникальным среди всех классов; к имени запроса не добавляется префикс в виде имени пакета или класса:

```
@NamedQueries({
    @NamedQuery(
        name = "findItemById",
        query = "select i from Item i where i.id = :id"
    )
})
@Entity
public class Item {
    // ...
}
```

Этот класс отмечен аннотацией `@NamedQueries`, содержащей массив аннотаций `@NamedQuery`. Единственный запрос может быть объявлен без дополнительной аннотации `@NamedQueries`. Для определения SQL-запросов вместо JPQL применяйте аннотацию `@NamedNativeQuery`. Для отображения результатов запроса SQL существует множество вариантов, и мы покажем их позже в главе 17.

Особенности Hibernate

К сожалению, аннотации именованных запросов JPA обязательно должны находиться перед отображаемым классом. Их нельзя поместить в файл метаданных package-info.java. Для этих целей в Hibernate есть собственные аннотации:

Файл: /model/src/main/java/org/jpwh/model/querying/package-info.java

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByName",
        query = "select i from Item i order by i.name asc"
    )
})

package org.jpwh.model.querying;
```

Если для хранения именованных запросов вас не устраивают ни файлы XML, ни аннотации, можно создавать их программно.

14.4.4. Программное создание именованных запросов

Объект `Query` можно «превратить» в именованный запрос с помощью метода `EntityManagerFactory#addNamedQuery()`:

```
Query findItemsQuery = em.createQuery("select i from Item i");
em.getEntityManagerFactory().addNamedQuery(
```

```
    "savedFindItemsQuery", findItemsQuery
);
Query query =
                                                  Последующий вызов
                                                  с тем же экземпляром
                                                  EntityManagerFactory
em.createNamedQuery("savedFindItemsQuery");
```

Этот код регистрирует запрос в единице хранения – экземпляре `EntityManagerFactory` – и обеспечивает повторное использование именованного запроса. Сохраняемый объект `Query` не обязательно должен быть выражением JPQL; можно сохранять запросы на основе критериев и обычные SQL-запросы. Обычно запрос регистрируется только один раз, во время запуска приложения.

Использовать или не использовать именованные запросы – решать вам. Но вообще хранение строк запросов в коде приложения (если только они не размещены в аннотациях) мы рассматриваем во вторую очередь; при любой возможности избавляйтесь от строк запросов в программном коде. На практике самым универсальным способом являются файлы XML.

Кроме того, для некоторых запросов может понадобиться применить дополнительные настройки и подсказки.

14.5. Подсказки для запросов

В этом разделе мы представим несколько дополнительных параметров из стандарта JPA, а также несколько настроек, поддерживаемых Hibernate. Название раздела подразумевает, что данная информация может понадобиться не сразу, поэтому вы можете пропустить этот раздел и обратиться к нему позже, когда это потребуется.

Во всех примерах используется один и тот же запрос:

```
String queryString = "select i from Item i";
```

Фактически подсказку можно включить в запрос (`Query`) с помощью метода `setHint()`. Остальные API для работы с запросами, такие как `TypedQuery` или `StoredProcedureQuery`, также имеют этот метод. Если реализация механизма хранения не поддерживает подсказки, она просто проигнорирует их.

Следующие подсказки, перечисленные в табл. 14.1, стандартизованы в JPA.

Таблица 14.1. Подсказки для запросов, стандартизованные в JPA

Имя	Значение	Описание
`javax.persistence.query.timeout`	(Миллисекунды)	Устанавливает предельное время выполнения запроса. Доступна также в виде константы `org.hibernate.annotations.QueryHints.TIMEOUT_JPA`
`javax.persistence.cache.retrieveMode`	`USE | BYPASS`	Указывает Hibernate, следует ли обращаться к разделяемому кэшу второго уровня при извлечении данных, или данные должны читаться только из результата запроса
`javax.persistence.cache.storeMode`	`USE | BYPASS | REFRESH`	Указывает Hibernate, следует ли сохранять данные в кэше второго уровня при извлечении их из результатов запроса

Особенности Hibernate

В Hibernate имеются собственные подсказки для запросов, доступные в виде констант в классе `org.hibernate.annotations.QueryHints`; см. табл. 14.2.

Таблица 14.2. Подсказки для запросов в Hibernate

Имя	Значение	Описание
`org.hibernate.flushMode`	`org.hibernate.FlushMode` (перечисление)	Определяет необходимость выталкивания контекста хранения перед выполнением запроса
`org.hibernate.readOnly`	`true` | `false`	Включает или отключает проверку состояния управляемых экземпляров сущностей, возвращаемых запросом
`org.hibernate.fetchSize`	(Количество одновременно извлекаемых записей в JDBC)	Перед выполнением запроса вызывает метод JDBC `PreparedStatement#setFetchSize()`; это оптимизационная подсказка для драйвера базы данных
`org.hibernate.comment`	(Комментарий для кода SQL)	Комментарий, добавляемый перед кодом SQL; используется для журналирования (в том числе на стороне базы данных)

Кэш второго уровня (особенно для запросов) – сложная тема, потому мы отдельно посвятим ей раздел 20.2. Вы должны будете прочитать этот раздел, прежде чем использовать разделяемый кэш: установка флага кэширования запроса в значение «включено» не окажет никакого эффекта.

Некоторые подсказки заслуживают более подробных пояснений.

14.5.1. Установка предельного времени выполнения

Время выполнения запроса можно контролировать, задав *предельное время выполнения*:

```
Query query = em.createQuery(queryString)
    .setHint("javax.persistence.query.timeout", 60000);  ← 1 минута
```

В Hibernate этот метод обладает такой же семантикой и влиянием, как и `setQueryTimeout()` JDBC-интерфейса `Statement`.

Обратите внимание, что драйвер JDBC не обязательно остановит запрос при истечении времени ожидания. Спецификация JDBC гласит: «Как только у источника данных появится возможность обработать запрос на прерывание текущей команды, пользователю будет возвращено исключение `SQLException`...» Это позволяет достаточно широко интерпретировать момент, когда у источника данных появится возможность прервать выполнение команды. Это может быть и после ее завершения. Вы можете проверить это со своей СУБД и драйвером. Для этого установите подсказку как глобальное свойство в файле persistence.xml, передайте ее как свойство при создании экземпляра `EntityManagerFactory` или используйте ее

как параметр именованного запроса. Метод `Query#setHint()` переопределяет глобальное свойство для конкретного запроса.

14.5.2. Установка режима выталкивания контекста хранения

Предположим, что вы модифицировали несколько хранимых экземпляров сущностей перед выполнением запроса. Например, поменяли значение свойства `name` управляемых экземпляров `Item`. Эти изменения присутствуют только в памяти, поэтому перед выполнением запроса Hibernate по умолчанию *выталкивает* контекст хранения вместе со всеми изменениями в базу данных. Это гарантирует получение последних изменений запросом и отсутствие конфликта между результатами запроса и экземплярами в памяти.

Иногда это очень непрактично, когда выполняется последовательность попеременных изменений и обращений к базе и каждый следующий запрос извлекает другие наборы данных, отличные от предыдущих. Другими словами, может быть заранее известно, что перед выполнением запроса не понадобится выталкивать изменения в базу данных, поскольку конфликтующих результатов не будет. Также обратите внимание, что контекст хранения обеспечивает повторимое чтение экземпляров сущностей, поэтому проблему будут представлять лишь скалярные результаты.

Отключить выталкивание контекста хранения перед выполнением запроса можно с помощью подсказки `org.hibernate.flushMode` для объекта `Query` со значением `org.hibernate.FlushMode.COMMIT`. К счастью, в JPA определен стандартный метод `setFlushMode()` интерфейсов `EntityManager` и `Query`; значение `FlushModeType.COMMIT` также стандартизовано. Поэтому, если требуется отключить выталкивание контекста для конкретного запроса, используйте стандартный API:

```
Query query = em.createQuery(queryString)
    .setFlushMode(FlushModeType.COMMIT);
```

В режиме `COMMIT` Hibernate не будет выталкивать контекст перед выполнением запроса. Значение по умолчанию – `AUTO`.

14.5.3. Установка режима только для чтения

В разделе 10.2.8 мы рассказывали, как уменьшить потребление памяти и избежать длинных циклов проверки состояния объектов. Используя подсказку, вы можете сообщить Hibernate, что все экземпляры сущностей, возвращаемые запросом, следует рассматривать как доступные только для чтения (хотя и неотсоединенные):

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.READ_ONLY,
        true
    );
```

Все экземпляры `Item`, возвращаемые этим запросом, находятся в хранимом состоянии, но Hibernate не будет делать снимков состояния для автоматической проверки объектов в контексте хранения. Hibernate не сохранит никаких изме-

нений, если только вы не отключите описываемого режима вызовом `session.setReadOnly(item, false)`.

14.5.4. Определение количества одновременно извлекаемых записей

Количество одновременно извлекаемых записей – это подсказка для драйвера базы данных:

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.FETCH_SIZE,
        50
    );
```

Если эта возможность не поддерживается драйвером, данная подсказка может никак не повлиять на производительность. Но если поддерживается, она может ускорить взаимодействие между клиентом JDBC и базой данных: при обработке результата запроса (`ResultSet`) клиентом (Hibernate) за одно обращение будет извлекаться сразу несколько записей.

14.5.5. Управление комментариями SQL

Во время оптимизации приложения часто приходится читать множество сообщений из журнала выполнения SQL. Мы настоятельно рекомендуем добавить параметр `hibernate.use_sql_comments` в файл конфигурации persistence.xml. В этом случае Hibernate будет добавлять автоматически сформированные комментарии к каждому выражению SQL, сохраняемому в журнале выполнения.

Добавить собственный комментарий к конкретному запросу (`Query`) можно с помощью рекомендации:

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.COMMENT,
        "Custom SQL comment"
    );
```

Все предыдущие подсказки были связаны с Hibernate или с JDBC. Но для большинства разработчиков и администраторов баз данных подсказки для запросов представляют собой нечто совершенно иное. В SQL подсказка для запроса – это инструкция в SQL-выражении для оптимизатора СУБД. Например, если разработчик или администратор решит, что план выполнения, выбранный оптимизатором базы данных для конкретного выражения SQL, недостаточно быстрый, он сможет использовать подсказку для выбора другого плана выполнения. Hibernate и спецификация Java Persistence не поддерживают произвольных подсказок SQL на уровне API; для этого придется использовать чистый SQL и создавать собственные выражения SQL; выполнять такие запросы вы сможете, используя поддерживаемые API.

С другой стороны, в некоторых СУБД можно управлять оптимизатором с помощью SQL-комментария, расположенного в начале выражения SQL. В таком случае можно использовать такую же подсказку, как в предыдущем примере.

Вы видели, как можно установить подсказку напрямую в экземпляре `Query`. Если вы храните запросы во внешних файлах и используете именованные запросы, подсказки нужно определять либо в аннотациях, либо в XML.

14.5.6. Подсказки для именованных запросов

Все подсказки для запросов, которые устанавливались выше методом `setHint()`, также можно указать в метаданных XML, в элементах `<named-query>` или `<named-native-query>`:

```
<entity-mappings
   version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
            http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

    <named-query name="findItems">
        <query><![CDATA[
            select i from Item i
        ]]></query>
        <hint name="javax.persistence.query.timeout" value="60000"/>
        <hint name="org.hibernate.comment" value="Custom SQL comment"/>
    </named-query>

</entity-mappings>
```

Аналогично можно задавать подсказки в определениях именованных запросов в аннотациях:

```
@NamedQueries({
    @NamedQuery(
        name = "findItemByName",
        query = "select i from Item i where i.name like :name",
        hints = {
            @QueryHint(
                name = org.hibernate.annotations.QueryHints.TIMEOUT_JPA,
                value = "60000"),
            @QueryHint(
                name = org.hibernate.annotations.QueryHints.COMMENT,
                value = "Custom SQL comment")
        }
    )
})
```

Особенности Hibernate

Подсказки для именованных запросов также можно задавать с помощью аннотаций Hibernate, в файле package-info.java:

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemBuyNowPriceGreaterThan",
        query = "select i from Item i where i.buyNowPrice > :price",
        timeout = 60,
        comment = "Custom SQL comment"
    )
})

package org.jpwh.model.querying;
```

Особенности Hibernate

И наконец, подсказки можно определять для запросов в XML-файлах конфигурации Hibernate:

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <query name="findItemsOrderByAuctionEndHibernateWithHints"
           cache-mode="ignore"
           comment="Custom SQL comment"
           fetch-size="50"
           read-only="true"
           timeout="60">
        select i from Item i order by i.auctionEnd asc
    </query>
</hibernate-mapping>
```

14.6. Резюме

- Вы узнали, как создавать и выполнять запросы. Для создания и обработки результатов запросов используются интерфейсы JPA. Вы также познакомились с интерфейсами запросов Hibernate.
- Вы узнали, как готовить запросы к выполнению и защищаться от атак, связанных с внедрением кода SQL. Вы познакомились со связанными и позиционными параметрами, а также с постраничной обработкой больших результатов запросов.

- Чтобы не встраивать JPQL-запросов в исходные файлы на Java, можно присвоить им имена и переместить во внешние файлы. Вы видели, как вызывать именованные запросы, а также узнали способы их определения: в метаданных XML, в аннотациях и программно.
- Мы рассказали о подсказках для запросов, которые можно передавать в Hibernate: предельное время выполнения, доступ только на чтение, количество одновременно извлекаемых записей и комментарии SQL. На примере JPQL мы продемонстрировали, как задавать подсказки для запросов во внешних файлах.

Глава 15

Языки запросов

В этой главе:
- запросы JPQL и запросы на основе критериев;
- эффективное извлечение данных с помощью соединений;
- запросы и подзапросы для получения отчетов.

Создание запросов – наиболее интересный этап в разработке приложений баз данных. Для создания сложного запроса может потребоваться много времени, и он может оказать колоссальное влияние на производительность приложения. С другой стороны, писать запросы становится проще по мере приобретения опыта, и то, что сначала казалось сложным, становится простым после изучения различных *языков запросов*.

В этой главе рассматриваются языки запросов, доступные в JPA: JPQL и API для запросов на основе критериев. Мы будем показывать примеры запросов, созданных с использованием обоих языков/API и возвращающих одинаковые результаты.

Главные нововведения в JPA 2

- Добавлена поддержка операторов `CASE`, `NULLIF` и `COALESCE`, обладающих такой же семантикой, как и их аналоги в SQL.
- В выборке и ограничении запроса можно выполнить приведение к более специфичному типу с помощью оператора `TREAT`.
- В ограничениях и проекциях запросов можно вызывать произвольные SQL-функции.
- Используя новое ключевое слово `ON`, можно добавлять дополнительные условия соединений.
- В предложениях `FROM` подзапросов можно использовать соединения.

Мы рассчитываем, что во время разработки приложения вы будете неоднократно возвращаться к этой главе, используя ее в качестве руководства по синтаксису запросов. Вследствие этого наше изложение будет кратким и сопровождаться небольшими примерами кода для различных вариантов использования. Иногда мы будем упрощать части приложения CaveatEmptor ради удобочитаемости. Напри-

мер, в сравнениях мы будем обращаться не к экземплярам `MonetaryAmount`, а к экземплярам `BigDecimal`.

Для начала разберемся с терминологией, относящейся к запросам. Для определения источников данных используются *выборки* (selections), для отбора записей, удовлетворяющих определенным критериям, – *ограничения* (restrictions), и для перечисления выбираемых столбцов данных – *проекции* (projections). Вы увидите, что данная глава организована в том же порядке.

В этой главе, говоря о запросах, мы, как правило, будем иметь в виду выражения `SELECT`: операции, извлекающие информацию из базы данных. JPA также поддерживает выражения `UPDATE`, `DELETE`, выражение `INSERT ... SELECT` в JPQL, запросы на основе критериев и различные реализации SQL, о которых мы расскажем в разделе 20.1. Мы не будем повторять здесь этих массовых операций и сосредоточимся только на выражениях `SELECT`. Для начала покажем несколько простых примеров выборок.

15.1. Выборка

Во-первых, говоря *выборка*, мы не имеем в виду предложения запроса `SELECT`. Мы также не имеем в виду самого выражения `SELECT`. Мы говорим о *выборе реляционной переменной*, или, говоря на языке SQL, о предложении `FROM`. Оно описывает источник данных, или, проще говоря, таблицы, из которых будет производиться выборка. Используя имена классов вместо таблиц, в JPQL можно написать следующее:

```
from Item
```

Следующий запрос (только предложение `FROM`) извлекает все экземпляры `Item`. Для него Hibernate сгенерирует такой SQL-запрос:

```
select i.ID, i.NAME, ... from ITEM i
```

Эквивалентный запрос на основе критериев можно создать, передав имя сущности в метод `from()`:

```
CriteriaQuery criteria = cb.createQuery(Item.class);
criteria.from(Item.class);
```

Особенности Hibernate

Hibernate может работать с запросами, содержащими лишь одно предложение `FROM` или критерий. К сожалению, ни JPQL, ни запросы на основе критериев, которые мы только что продемонстрировали, не являются переносимыми: они несовместимы с JPA. Спецификация JPA требует, чтобы запрос JPQL содержал предложение `SELECT`, а переносимые запросы на основе критериев вызывали метод `select()`.

Это требует назначения псевдонимов и определения корневых источников запроса, что является нашей следующей темой.

15.1.1. Назначение псевдонимов и определение корневых источников запроса

Добавление предложения SELECT в запрос JPQL требует присваивания *псевдонима* (alias) классу в предложении FROM, чтобы можно было ссылаться на него в других предложениях запроса.

```
select i from Item as i
```

После этого следующий запрос будет совместим с JPA. Как обычно, ключевое слово as является необязательным. Следующий запрос аналогичен предыдущему:

```
select i from Item i
```

Извлекаемым экземплярам класса Item присваивается псевдоним i. Это похоже на объявление временной переменной в следующем коде на Java:

```
for(Iterator i = result.iterator(); i.hasNext();) {
    Item item = (Item) i.next();
    // ...
}
```

Псевдонимы в запросах нечувствительны к регистру, поэтому запрос select iTm from Item itm также будет работать. Мы предпочитаем использовать короткие и простые псевдонимы; они должны быть уникальными в пределах запроса (или подзапроса).

Переносимые запросы на основе критериев должны вызывать метод select():

```
CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);
criteria.select(i);
```

В большинстве примеров с использованием запросов на основе критериев мы будем убирать строку cb.createQuery(); она всегда выглядит одинаково. Каждый раз, встречая переменную criteria, знайте, что она получена с помощью вызова CriteriaBuilder#createQuery(). Как получить экземпляр CriteriaBuilder, рассказывалось в предыдущей главе.

Экземпляр Root всегда ссылается на сущность. Позже вы увидите запросы с несколькими корневыми элементами. Этот запрос можно записать короче, если не создавать дополнительной переменной типа Root:

```
criteria.select(criteria.from(Item.class));
```

Тип сущности также можно указать динамически, с помощью Metamodel API:

```
EntityType entityType = getEntityType(
    em.getMetamodel(), "Item"
);

criteria.select(criteria.from(entityType));
```

Мы написали метод `getEntityType()` исключительно для удобства: он обходит все экземпляры коллекции `Metamodel#getEntities()` и находит сущность с соответствующим именем.

Сущность `Item` не имеет подклассов, поэтому в следующем разделе мы рассмотрим полиморфные запросы.

15.1.2. Полиморфные запросы

JPQL, как и подобает объектно-ориентированному языку запросов, поддерживает *полиморфные запросы*, извлекающие не только экземпляры класса, но и все экземпляры подклассов. Рассмотрим следующие запросы:

```
select bd from BillingDetails bd
```

```
criteria.select(criteria.from(BillingDetails.class));
```

Они возвращают все экземпляры типа `BillingDetails`, который сам является абстрактным классом. В этом случае каждый экземпляр будет относиться к одному из подтипов `BillingDetails`: `CreditCard` или `BankAccount`. Чтобы получить только экземпляры конкретного подкласса, можно написать следующий запрос:

```
select cc from CreditCard cc
```

```
criteria.select(criteria.from(CreditCard.class));
```

Класс, указанный в предложении `FROM`, не обязан быть отображаемым хранимым классом; подойдет любой класс. Следующий запрос возвращает все сохраненные объекты:

```
select o from java.lang.Object o
```

Особенности Hibernate

Да, вы можете выбрать записи из всех таблиц базы данных и загрузить их в память! Это работает и с интерфейсами – можно, к примеру, выбрать все сериализуемые экземпляры:

```
select s from java.io.Serializable s
```

Плохая новость в том, что JPA не стандартизует полиморфных запросов JPQL с использованием интерфейсов. Они работают только в Hibernate, но переносимое приложение должно ссылаться в предложении `FROM` лишь на отображаемые классы сущностей, такие как `BillingDetails` или `CreditCard`. Метод `from()` из API запросов на основе критериев принимает только типы отображаемых сущностей.

Можно выполнить *неполиморфный* запрос, ограничив набор возвращаемых типов с помощью функции `TYPE`. Чтобы извлечь только экземпляры конкретного подкласса, можно выполнить такой запрос:

```
select bd from BillingDetails bd where type(bd) = CreditCard
```

```
Root<BillingDetails> bd = criteria.from(BillingDetails.class);
criteria.select(bd).where(
    cb.equal(bd.type(), CreditCard.class)
);
```

Для параметризации этого запроса нужно добавить предложение `IN` с именованным параметром:

```
select bd from BillingDetails bd where type(bd) in :types
```

```
Root<BillingDetails> bd = criteria.from(BillingDetails.class);
criteria.select(bd).where(
    bd.type().in(cb.parameter(List.class, "types"))
);
```

Связывание аргумента с параметром запроса происходит во время передачи списка (`List`) возвращаемых типов:

```
Query query = // ...
query.setParameter("types", Arrays.asList(CreditCard.class,
    BankAccount.class));
```

Чтобы выбрать все экземпляры подкласса, *кроме* конкретного класса, можно выполнить следующий запрос:

```
select bd from BillingDetails bd where not type(bd) = BankAccount
```

```
Root<BillingDetails> bd = criteria.from(BillingDetails.class);
criteria.select(bd).where(
    cb.not(cb.equal(bd.type(), BankAccount.class))
);
```

Полиморфизм в запросах применяется не только к конкретным классам, но и к полиморфным ассоциациям, как вы увидите далее.

На этом мы заканчиваем описание первого этапа создания запросов – определение *выборки*. Мы определили таблицы, из которых будем извлекать данные. Далее вам может понадобиться уменьшить количество извлекаемых записей с помощью *ограничения*.

15.2. Ограничения

Как правило, извлекать все экземпляры класса из базы данных не требуется. Поэтому вы должны уметь описывать критерии отбора данных, возвращаемых запросом. Такие описания мы называем *ограничениями*. В SQL и JPQL ограничения описываются с помощью предложения `WHERE`, а в API на основе критериев аналогичную функцию выполняет метод `where()`.

Ниже показан пример типичного предложения WHERE, ограничивающего набор результатов только экземплярами Item с конкретным именем.

```
select i from Item i where i.name = 'Foo'
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.equal(i.get("name"), "Foo")
);
```

Ограничение в запросе выражается в терминах поля name класса Item.

Ниже показан код SQL, сгенерированный данным запросом:

```
select i.ID, i.NAME, ... from ITEM i where i.NAME = 'Foo'
```

В выражениях и условиях можно использовать строковые литералы, заключая их в одиночные кавычки. Для литералов, представляющих дату, время и метки времени, нужно использовать синтаксис экранирования JDBC: `... where i.auctionEnd = {d '2013-26-06'}`. Обратите внимание, что способ обработки такого литерала, а также различные варианты его написания определяются драйвером JDBC и СУБД. И не забудьте совет из предыдущей главы: не внедряйте непроверенных входных данных пользователя в строку запроса – используйте параметры. В JDBC также часто встречаются литералы true и false:

```
select u from User u where u.activated = true
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(
    cb.equal(u.get("activated"), true)
);
```

В SQL (а также в JPQL и запросах на основе критериев) ограничения описываются с помощью троичной логики. Предложение WHERE – это логическое выражение, результатом которого может быть true, false или null.

Что такое троичная логика?

Запись будет добавлена в результат SQL-запроса тогда и только тогда, когда выражение в предложении WHERE вернет true. В языке Java результатом выражения nonNullObject == null является false, а null == null – true. В языке SQL оба выражения – NOT_NULL_COLUMN = null и null = null – вернут null, а не true. Поэтому для проверки выражения на равенство null в SQL приходится использовать дополнительные операторы IS NULL и IS NOT NULL. *Троичная логика* нужна для работы с выражениями со столбцами, которые могут содержать null. Интерпретируйте null не как специальную метку, а как обычное значение – это расширение знакомой двоичной логики реляционной модели в языке SQL. Hibernate поддерживает троичную логику с помощью троичных операторов как в JPQL, так и в запросах на основе критериев.

Рассмотрим наиболее распространенные операторы сравнения в логических выражениях, включая троичные операторы.

15.2.1. Выражения сравнения

JQPL и API запросов на основе критериев поддерживают такой же набор операторов сравнения, как и SQL. Ниже представлено несколько примеров, которые должны быть знакомы всем, кто знает язык SQL.

Следующий запрос возвращает все ставки с ценой в заданном диапазоне:

```
select b from Bid b where b.amount between 99 and 110
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.between(
        b.<BigDecimal>get("amount"),   ← Нужно указать тип атрибута
        new BigDecimal("99"), new BigDecimal("110")   ← Аргументы должны быть того же типа
    )
);
```

Запрос на основе критериев может показаться несколько странным; вероятно, вы нечасто видите параметризацию в середине выражений на Java. Метод `Root#get()` возвращает объект `Path<X>`, описывающий путь к атрибуту сущности. Для обеспечения типобезопасности требуется явно указать тип атрибута для этого пути `Path`, как в выражении `<BigDecimal>get("amount")`. Два оставшихся аргумента метода `between()` должны иметь тот же тип, иначе сравнение будет бессмысленным или вовсе не скомпилируется.

Следующий запрос возвращает все ставки с ценой выше заданной:

```
select b from Bid b where b.amount > 100
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.gt(   ← Метод gt() работает только с подклассами Number;
        b.<BigDecimal>get("amount"),   для других типов используйте метод greaterThan()
        new BigDecimal("100")
    )
);
```

Метод `gt()` принимает только аргументы подтипов `Number`, таких как `BigDecimal` или `Integer`. Чтобы сравнить значения других типов, например `Date`, используйте метод `greaterThan()`:

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.greaterThan(
        i.<Date>get("auctionEnd"),
        tomorrowDate
    )
);
```

Следующий запрос возвращает всех пользователей с именами «johndoe» и «janeroe».

```
select u from User u where u.username in ('johndoe', 'janeroe')
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(
    cb.<String>in(u.<String>get("username"))
        .value("johndoe")
        .value("janeroe")
);
```

В ограничениях с перечислениями следует указывать полное квалифицированное значение:

```
select i from Item i
    where i.auctionType = org.jpwh.model.querying.AuctionType.HIGHEST_BID
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.equal(
        i.<AuctionType>get("auctionType"),
        AuctionType.HIGHEST_BID
    )
);
```

Из-за того, что SQL использует троичную логику, проверка значений на равенство `null` требует специального подхода. JPQL предоставляет операторы `IS [NOT] NULL`, а в API запросов на основе критериев – методы `isNull()` и `isNotNull()`.

В следующем запросе `IS NULL` и `isNull()` используются для поиска товаров без текущей цены:

```
select i from Item i where i.buyNowPrice is null
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.isNull(i.get("buyNowPrice"))
);
```

Используя `IS NOT NULL` и `isNotNull()`, можно получить все товары с указанной текущей ценой:

```
select i from Item i where i.buyNowPrice is not null
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.isNotNull(i.get("buyNowPrice"))
);
```

Оператор `LIKE` позволяет организовать поиск, в котором часть шаблона может быть произвольной; как и в SQL, для этого применяются символы `%` и `_`:

```
select u from User u where u.username like 'john%'
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(
    cb.like(u.<String>get("username"), "john%")
);
```

Выражение `john%` истинно только для пользователей, имена которых начинаются со строки «john». Также с оператором `LIKE` можно использовать отрицание:

```
select u from User u where u.username not like 'john%'
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(
    cb.like(u.<String>get("username"), "john%").not()
);
```

Можно выполнить поиск всех вхождений подстроки, окружив искомую строку символами процента:

```
select u from User u where u.username like '%oe%'
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).where(
    cb.like(u.<String>get("username"), "%oe%")
);
```

Символ процента означает «произвольная последовательность символов»; подчеркивание – «произвольный символ». Если понадобится использовать литералы процента и подчеркивания, их можно экранировать любым символом:

```
select i from Item i
    where i.name like 'Name\_with\_underscores' escape :escapeChar
query.setParameter("escapeChar", "\\");
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.like(i.<String>get("name"), "Name\\_with\\_underscores", '\\')
);
```

Эти запросы найдут товар с именем `имя_с_подчеркиваниями`. В строках языка Java для экранирования используется символ \, который тоже нужно экранировать, поэтому в предыдущем примере этот символ удвоен.

JPA также поддерживает арифметические выражения:

```
select b from Bid b where (b.amount / 2) - 0.5 > 49
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.gt(
        cb.diff(
            cb.quot(b.<BigDecimal>get("amount"), 2),
            0.5
```

```
        ),
        49
    )
);
```

Выражения объединяются с помощью логических операторов (и группирующих скобок):

```
select i from Item i
    where (i.name like 'Fo%' and i.buyNowPrice is not null)
          or i.name = 'Bar'/
```

```
Root<Item> i = criteria.from(Item.class);

Predicate predicate = cb.and(
    cb.like(i.<String>get("name"), "Fo%"),
    cb.isNotNull(i.get("buyNowPrice"))
);

predicate = cb.or(
    predicate,
    cb.equal(i.<String>get("name"), "Bar")
);

criteria.select(i).where(predicate);
```

Особенности Hibernate

Если нужно объединить все предикаты с помощью логического И, для этого мы выбираем более удобный API запросов на основе критериев:

```
Root<Item> i = criteria.from(Item.class);

criteria.select(i).where(
    cb.like(i.<String>get("name"), "Fo%"),
    // и
    cb.isNotNull(i.get("buyNowPrice"))
    // и ...
);
```

Ниже, в табл. 15.1, перечислены все операторы в порядке убывания приоритета.

Таблица 15.1. Приоритет операторов JPQL

Оператор JPQL	API запросов на основе критериев	Описание
.	Н/А	Оператор в выражении, описывающем путь к атрибуту
+, -	neg()	Унарный плюс или минус (все беззнаковые числовые значения считаются положительными)

Таблица 15.1 *(окончание)*

Оператор JPQL	API запросов на основе критериев	Описание
*, /	prod(), quot()	Умножение и деление числовых выходых параметров
+, -	sum(), diff()	Сложение и вычитание числовых выходых параметров
=, <>, <, >, >=, <	equal(), notEqual(), lessThan(), lt(), greaterThan(), gt(), greaterThanEqual(), ge(), lessThan(), lt()	Операторы сравнения с семантикой SQL
[NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL	between(), like(), in(), isNull(), isNotNull()	Операторы сравнения с семантикой SQL
IS [NOT] EMPTY, [NOT]	isEmpty(), isNotEmpty(), isMember(), isNotMember()	Операторы для хранимых коллекций
MEMBER [OF] NOT, AND, OR	not(), and(), or()	Логические операторы для управления порядком вычислений

Вы уже знаете, что двухместные операторы сравнения обладают такой же семантикой, как их аналоги в SQL, и могут группироваться и упорядочиваться с помощью логических операторов. Давайте посмотрим на работу с коллекциями.

15.2.2. Выражения с коллекциями

В каждом выражении из предыдущего раздела были только пути к атрибутам, имеющим единственное значение: `user.username`, `item.buyNowPrice` и т. д. Но можно также написать выражение, указывающее на коллекцию, и применить к ней некоторые операторы и функции.

Предположим, к примеру, что требуется найти все категории – экземпляры `Catergory`, – имеющие хотя бы один товар в коллекции `items`:

```
select c from Category c
    where c.items is not empty
```

```
Root<Category> c = criteria.from(Category.class);
criteria.select(c).where(
    cb.isNotEmpty(c.<Collection>get("items"))
);
```

Результат выражения `c.items` в запросе JPQL ссылается на коллекцию `items` экземпляра `Category`. Обратите внимание, что нельзя обращаться к атрибутам элементов коллекции в подобном выражении: нельзя написать `c.items.buyNowPrice`.

Отфильтровать категории по количеству входящих в них товаров можно, используя функцию `size()`:

```
select c from Category c
    where size(c.items) > 1
```

```
Root<Category> c = criteria.from(Category.class);
```

```
criteria.select(c).where(
    cb.gt(
        cb.size(c.<Collection>get("items")),
        1
    )
);
```

Также можно найти категорию, содержащую конкретный товар:

```
select c from Category c
    where :item member of c.items
```

```
Root<Category> c = criteria.from(Category.class);
criteria.select(c).where(
    cb.isMember(
        cb.parameter(Item.class, "item"),
        c.<Collection<Item>>get("items")
    )
);
```

Для хранимых словарей определены дополнительные операторы `key()`, `value()` и `entry()`. Предположим, что каждый экземпляр `Item` (товар) имеет хранимый словарь встроенных объектов `Image` (изображений), как показано в разделе 7.2.4. Имена файлов изображений являются ключами словаря. Следующий запрос извлекает все экземпляры `Image` с именами, оканчивающимися на .jpg:

```
select value(img)
    from Item i join i.images img
    where key(img) like '%.jpg'
```

Оператор `value()` возвращает значения из словаря `Map`, а оператор `key()` – набор ключей. Для получения записей из словаря – экземпляров `Map.Entry` – используйте оператор `entry()`.

Рассмотрим далее оставшиеся функции, применимые не только к коллекциям.

15.2.3. Вызовы функций

Возможность вызова функций в предложении `WHERE` – одна из самых сильных сторон языков запросов. Следующие запросы вызывают функцию `lower()` для поиска подстроки без учета регистра:

```
select i from Item i where lower(i.name) like 'ba%'
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.like(cb.lower(i.<String>get("name")), "ba%")
);
```

Все доступные функции перечислены в табл. 15.2. Запросы на основе критериев имеют аналогичные методы в `CriteriaBuilder`, лишь немного отличающиеся способом записи имен (используется «верблюжийРегистр» без подчеркиваний).

Таблица 15.2. Функции запросов JPA (перегруженные методы не показаны)

Функция	Область применения
`upper(s), lower(s)`	Строковые значения; возвращает строку
`concat(s, s)`	Строковые значения; возвращает строку
`current_date, current_time, current_timestamp`	Возвращает дату и/или время на сервере управления базами данных
`substring(s, offset, length)`	Строковые значения (смещение начинается с 1); возвращает строку
`trim([[both|leading|trailing] char [from]] s)`	Удаляет пробелы с обеих (`both`) сторон строки `s`, если не указан иной символ `char` или не заданы другие правила; возвращает строку
`length(s)`	Строковое значение; возвращает число
`locate(search, s, offset)`	Возвращает позицию подстроки `search` в строке `s`, начиная поиск с позиции `offset`; возвращает числовое значение
`abs(n), sqrt(n), mod(dividend, divisor)`	Числовые значения; возвращает абсолютное значение того же типа, что и аргумент, квадратный корень как `Double` и остаток от деления как `Integer`
`treat(x as Type)`	Приведение типа к подтипу в ограничениях; например, когда нужно найти всех пользователей с кредитными картами, срок действия которых заканчивается в 2013: `select u from User u where treat(u.billingDetails as CreditCard).expYear = '2013'`. (Обратите внимание, что это необязательно делать в Hibernate. Он автоматически выполнит приведение типа к подтипу, если используется поле подкласса)
`size(c)`	Коллекция выражений; возвращает `Integer` или 0, когда коллекция пуста
`index(orderedCollection)`	Выражение, возвращающее коллекцию, отображаемую с помощью `@OrderColumn`; возвращает значение типа `Integer`, соответствующее позиции аргумента в коллекции. Например, запрос `select i.name from Category c join c.items i where index(i) = 0` вернет названия для каждого первого товара в каждой категории

Особенности Hibernate

Как показано в табл. 15.3, Hibernate поддерживает дополнительные функции для JPQL. Стандартный JPA API запросов на основе критериев не имеет аналогов этих функций.

Таблица 15.3. Функции запросов Hibernate

Функция	Описание
`bit_length(s)`	Возвращает количество бит в `s`
`second(d), minute(d), hour(d), day(d), month(d), year(d)`	Извлекает время и дату из аргумента, представляющего время
`minelement(c), maxelement(c), minindex(c), maxindex(c), elements(c), indices(c)`	Возвращает элемент коллекции или индекс для коллекций, поддерживающих индексирование (словарей, списков, массивов)
`str(x)`	Выполняет приведение аргумента к строковому типу

Большинство этих функций транслируется в соответствующие функции SQL, которые вы видели ранее. Также можно вызывать функции SQL, поддерживаемые вашей СУБД и не показанные здесь.

Особенности Hibernate

Любая функция, встреченная в предложении `WHERE` запроса JPQL и неизвестная Hibernate, передается напрямую в базу данных в виде вызова функции SQL. Следующий запрос, например, вернет товары, аукцион для которых длился больше одного дня:

```
select i from Item i
    where
        datediff('DAY', i.createdOn, i.auctionEnd)
    > 1
```

Здесь вызывается нестандартная функция `datediff()`, доступная в базе данных H2, возвращающая разницу в днях между датами создания товара и окончания аукциона для данного экземпляра `Item`. Такой синтаксис может использоваться только с Hibernate; в JPA для вызова произвольных функций SQL стандартизован следующий синтаксис:

```
select i from Item i
    where
        function('DATEDIFF', 'DAY', i.createdOn, i.auctionEnd)
    > 1
```

Первый аргумент функции `function()` – имя вызываемой функции SQL в одинарных кавычках. За ним должны следовать необходимые операнды, если имеются. Такой же запрос, но на основе критериев:

```
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(
    cb.gt(
        cb.function(
            "DATEDIFF",
            Integer.class,
            cb.literal("DAY"),
            i.get("createdOn"),
            i.get("auctionEnd")
        ),
        1
    )
);
```

Аргумент `Integer.class` определяет тип возвращаемого значения функции `datediff()`, но здесь он бесполезен, поскольку результат вызова функции в ограничении не возвращается.

Вызов функции в предложении SELECT передаст возвращаемое значение на сторону Java; в предложении SELECT также можно вызывать произвольные SQL-функции базы данных. Но, прежде чем говорить о *проекциях*, мы сначала узнаем, как упорядочивать результаты.

15.2.4. Упорядочение результатов запроса

В любом языке запросов есть механизм упорядочения результатов. В JPQL эту роль играет предложение ORDER BY, как в SQL.

Следующий запрос найдет всех пользователей и упорядочит их по именам (по умолчанию в порядке возврастания):

```
select u from User u order by u.username
```

Возрастание или убывание определяется ключевым словом asc или desc:

```
select u from User u order by u.username desc
```

В API запросов на основе критериев порядок сортировки должен указываться *обязательно*, с помощью функции asc() или desc():

```
Root<User> u = criteria.from(User.class);
criteria.select(u).orderBy(
    cb.desc(u.get("username"))
);
```

Упорядочивать можно по нескольким атрибутам:

```
select u from User u order by u.activated desc, u.username asc
```

```
Root<User> u = criteria.from(User.class);
criteria.select(u).orderBy(
    cb.desc(u.get("activated")),
    cb.asc(u.get("username"))
);
```

Упорядочение атрибутов со значением null

Если столбец, по которому производится упорядочение, может содержать NULL, записи с NULL могут оказаться в начале или в конце результатов. Это поведение определяется СУБД, поэтому для создания переносимого приложения всегда нужно указывать, должны ли записи с NULL находиться в конце или начале, добавив предложение ORDER BY ... NULLS FIRST|LAST. Hibernate поддерживает это предложение в JPQL, однако в JPA оно не стандартизовано. Вместо этого можно задать порядок по умолчанию, установив параметр конфигурации единицы хранения hibernate.order_by.default_null_ordering в значение none (по умолчанию), first или last.

Особенности Hibernate

Спецификация JPA позволяет использовать в предложении ORDER BY только те свойства/пути, которые присутствуют в предложении SELECT. Следующие запросы не переносимы, но зато работают в Hibernate:

```
select i.name from Item i order by i.buyNowPrice asc
```

```
select i from Item i order by i.seller.username desc
```

Будьте осторожнее с неявными внутренними соединениями в выражениях, представляющих пути, и предложениях ORDER BY: последний запрос вернет только экземпляры Item с заполненным полем seller. Это может оказаться неожиданным, поскольку тот же запрос, но без предложения ORDER BY, вернет все экземпляры Item (давайте на секунду забудем, что в нашей модели поле seller любого объекта Item всегда заполнено). Более подробное обсуждение внутренних соединений и выражений, описывающих пути, вы найдете в следующей главе.

Вы уже знаете, как писать предложения FROM, WHERE и ORDER BY. Вы умеете извлекать нужные сущности, применяя различные выражения для ограничения и упорядочивания результатов. Теперь осталось только определить набор возвращаемых атрибутов с помощью проекции.

15.3. Проекции

Говоря простым языком, выборки и ограничения в запросе определяют, из каких таблиц и какие записи будут извлекаться. *Проекция* же определяет, какие «столбцы» нужно вернуть приложению. В JPQL за проекцию отвечает предложение SELECT.

15.3.1. Проекция сущностей и скалярных значений

Рассмотрим, к примеру, следующие запросы:

```
select i, b from Item i, Bid b
```

```
Root<Item> i = criteria.from(Item.class);
Root<Bid> b = criteria.from(Bid.class);
criteria.select(cb.tuple(i, b));
/* Удобная альтернатива:
criteria.multiselect(
    criteria.from(Item.class),
    criteria.from(Bid.class)
);
*/
```

Как упоминалось ранее, с помощью такого запроса на основе критериев можно выбирать экземпляры разных сущностей, представленных объектами `Root`, вызывая метод `from()` несколько раз. Для добавления в проекцию нескольких элементов нужно вызвать метод `tuple()` объекта `CriteriaBuilder` или более удобный метод `multiselect()`.

Ниже создается декартово произведение всех экземпляров `Item` и `Bid`. Запросы возвращают упорядоченные пары экземпляров сущностей `Item` и `Bid`:

```
List<Object[]> result = query.getResultList();   ← ❶ Возвращает коллекцию List с элементами типа Object[]

Set<Item> items = new HashSet();
Set<Bid> bids = new HashSet();

for (Object[] row : result) {
    assertTrue(row[0] instanceof Item);   ← ❷ Индекс 0
    items.add((Item) row[0]);

    assertTrue(row[1] instanceof Bid);    ← ❸ Индекс 1
    bids.add((Bid)row[1]);
}

assertEquals(items.size(), 3);
assertEquals(bids.size(), 4);
assertEquals(result.size(), 12);   ← Декартово произведение
```

Запрос вернет коллекцию (`List`) с элементами типа `Object[]` ❶. Элемент с индексом 0 будет хранить объект `Item` ❷, а элемент с индексом 1 – объект `Bid` ❸.

Будучи произведением, результат содержит все возможные комбинации записей из таблиц, соответствующих сущностям `Item` и `Bid`. Очевидно, что в таком запросе нет практического смысла, но вас не должна удивлять возможность получения коллекции элементов `Object[]` в качестве результата запроса. Hibernate управляет всеми экземплярами сущностей `Item` и `Bid`, которые находятся в хранимом состоянии, в контекст хранения. Также обратите внимание, что два множества `HashSet` устраняют повторяющиеся экземпляры `Item` и `Bid`.

В API запросов на основе критериев имеется также возможность получения типизированного списка результатов с помощью интерфейса `Tuple`. Для этого сначала нужно создать объект типа `CriteriaQuery<Tuple>`, вызвав метод `createTupleQuery()`. Затем завершить определение запроса, присвоив классам сущностей псевдонимы:

```
CriteriaQuery<Tuple> criteria = cb.createTupleQuery();

// Или: CriteriaQuery<Tuple> criteria = cb.createQuery(Tuple.class);

criteria.multiselect(
    criteria.from(Item.class).alias("i"),   ← Задавать псевдонимы необязательно
    criteria.from(Bid.class).alias("b")
);

TypedQuery<Tuple> query = em.createQuery(criteria);
List<Tuple> result = query.getResultList();
```

Интерфейс `Tuple` поддерживает несколько способов обращения к результатам: по индексу, по псевдониму или с использованием нетипизированного доступа через метаданные:

```
for (Tuple tuple : result) {
    Item item = tuple.get(0, Item.class);   ← По индексу
    Bid bid = tuple.get(1, Bid.class);

    item = tuple.get("i", Item.class);   ← По псевдониму
    bid = tuple.get("b", Bid.class);

    for (TupleElement<?> element : tuple.getElements()) {   ← Через метаданные
        Class clazz = element.getJavaType();
        String alias = element.getAlias();
        Object value = tuple.get(element);
    }
}
```

Следующая проекция также возвращает коллекцию элементов `Object[]`:

```
select u.id, u.username, u.homeAddress from User u
```

```
Root<User> u = criteria.from(User.class);
criteria.multiselect(                ← Возвращает коллекцию List элементов Object[]
    u.get("id"), u.get("username"), u.get("homeAddress")
);
```

Каждый элемент `Object[]`, возвращаемый этим запросом, содержит объект типа `Long` по индексу `0`, объект `String` по индексу `1` и объект `Address` по индексу `2`. Первые два – скалярные значения; третий – экземпляр встроенного класса. Ни один из этих объектов не является управляемым экземпляром сущности! Следовательно, ни один не находится в хранимом состоянии, в отличие от экземпляров сущностей. Они не пересекают границу транзакции, и, очевидно, изменение их состояния не проверяется автоматически. Мы называем такие объекты *временными* (transient). Подобный запрос обычно требуется для создания простого отчета, который бы показывал все имена пользователей и их домашние адреса.

Вы уже несколько раз сталкивались с *выражениями для описания путей к атрибутам*: с помощью точечной нотации можно ссылаться на такие поля сущностей, как `User#username` в виде `u.username`. Для вложенного поля встраиваемого типа, к примеру, можно указать путь `u.homeAddress.city.zipcode`. Этот путь ссылается на единственное значение, поскольку не завершается именем поля с отображаемой коллекцией.

По сравнению с использованием объектов `Object[]` или `Tuple` (особенно для создания отчетов), лучше использовать способ динамического создания экземпляров в проекциях, который будет продемонстрирован далее.

15.3.2. Динамическое создание экземпляров

Предположим, что существует отчет, в котором нужно отобразить некоторые данные в виде списка. Допустим, нужно показать все аукционные товары с датами

окончания каждого аукциона. При этом не хочется загружать управляемые экземпляры сущности `Item`, поскольку никакие данные меняться не будут.

Сначала нужно создать класс `ItemSummary` с конструктором, принимающим аргументы `Long`, `String` и `Date`, соответствующие идентификатору товара, его названию и отметке времени окончания аукциона:

```
public class ItemSummary {

    public ItemSummary(Long itemId, String name, Date auctionEnd) {
        // ...
    }

    // ...
}
```

Иногда экземпляры таких классов называются *объектами передачи данных* (Data Transfer Objects, DTO), поскольку их главной целью является передача данных из одной части приложения в другую. Класс `ItemSummary` не отображается в базу данных, и в нем можно определять любые методы (чтения, записи, вывода значений), необходимые для вашего отчета.

Hibernate может сразу вернуть из запроса новые экземпляры `ItemSummary` с помощью оператора `new` в JPQL и метода `construct()` в критериях:

```
select new org.jpwh.model.querying.ItemSummary(
    i.id, i.name, i.auctionEnd
) from Item i
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(
    cb.construct(
        ItemSummary.class,                ← Должен быть подходящий конструктор
        i.get("id"), i.get("name"), i.get("auctionEnd")
    )
);
```

Каждый элемент в списке результатов этого запроса будет экземпляром `ItemSummary`. Обратите внимание, что в JPQL нужно указывать полное квалифицированное имя класса, т. е. добавлять имя пакета. Также отметьте, что использование вложенных конструкторов не поддерживается – нельзя написать `new ItemSummary(..., new UserSummary(...))`.

Динамическое создание экземпляров работает не только с неуправляемыми объектами передачи данных вроде экземпляров `ItemSummary`. Можно создать новый объект `Item` или `User`, уже являющийся экземпляром отображаемого класса сущности. Самое главное, чтобы класс имел подходящий конструктор для использования в проекции. Но при динамическом создании экземпляров сущностей в запросе они не будут находиться в хранимом состоянии! Они будут возвращаться либо во *временном*, либо в *отсоединенном* состоянии, в зависимости от значения идентификатора. Как вариант этот прием можно использовать для копирования данных: извлечь «новый» временный экземпляр `Item`, в конструктор которого

передается часть значений из базы данных, в часть из приложения, чтобы затем сохранить его в базу, вызвав метод `persist()`.

Если класс объекта DTO не имеет подходящего конструктора, но вы хотите заполнить свойства объекта значениями из результата запроса, используйте `ResultTransformer`, как показано в разделе 16.1.3. Другие примеры группировки и агрегирования вы увидите далее.

А пока рассмотрим одну особенность проекций, которая сбивает с толку многих разработчиков, – обработку повторяющихся записей.

15.3.3. Извлечение уникальных результатов

Проекция в запросе не гарантирует уникальности элементов в наборе результатов. Например, имена товаров могут быть неуникальными, поэтому следующий запрос может вернуть одно имя несколько раз:

```
select i.name from Item i
```

```
CriteriaQuery<String> criteria = cb.createQuery(String.class);
criteria.select(
    criteria.from(Item.class).<String>get("name")
);
```

Трудно представить, какую пользу могли бы принести две одинаковые записи в результате, поэтому если есть вероятность появления одинаковых записей, для их фильтрации обычно используется ключевое слово `DISTINCT` или метод `distinct()`:

```
select distinct i.name from Item i
```

```
CriteriaQuery<String> criteria = cb.createQuery(String.class);
criteria.select(
    criteria.from(Item.class).<String>get("name")
);
criteria.distinct(true);
```

Они удалят повторяющиеся значения из возвращаемого списка имен экземпляров `Item` и будут преобразованы в SQL-оператор `DISTINCT`. Фильтрация осуществляется на уровне базы данных. Далее вы увидите, что так происходит не всегда.

Вы уже видели, как вызывать функции в ограничениях, в предложении `WHERE`. Аналогично можно вызывать функции в проекциях для изменения возвращаемых результатов.

15.3.4. Вызов функций в проекциях

Следующие запросы возвращают строки (`String`), полученные путем вызова функции `concat()` внутри проекции:

```
select concat(concat(i.name, ': '), i.auctionEnd) from Item i
```

```
Root<Item> i = criteria.from(Item.class);
criteria.select(
```

```
    cb.concat(
        cb.concat(i.<String>get("name"), ":"),
        i.<String>get("auctionEnd")   ← Обратите внимание на приведение типа Date к строке
    )
);
```

Запрос вернет список (`List`) строк (`String`), каждая из которых будет иметь вид «[имя товара]:[дата окончания аукциона]». Из примера видно, что можно также описывать вложенные вызовы функций.

Функция `coalesce()` возвращает `null`, если все ее аргументы равны `null`; в противном случае она вернет первый непустой аргумент:

```
select i.name, coalesce(i.buyNowPrice, 0) from Item i
```

```
Root<Item> i = criteria.from(Item.class);
criteria.multiselect(
    i.get("name"),
    cb.coalesce(i.<BigDecimal>get("buyNowPrice"), 0)
);
```

Если товар (`Item`) не имеет текущей цены покупки (`buyNowPrice`), вместо `null` будет возвращено нулевое значение типа `BigDecimal`.

Похожим образом действует выражение `case/when`, но его возможности гораздо шире. Следующий запрос вернет имя (`username`) каждого пользователя (`User`), а также строку (`String`) с текстом «Германия», «Швейцария» или «Другое» в зависимости от длины индекса (`zipcode`) пользователя:

```
select
    u.username,
    case when length(u.homeAddress.zipcode) = 5 then 'Germany'
        when length(u.homeAddress.zipcode) = 4 then 'Switzerland'
        else 'Other'
    end
from User u
```

```
// Проверьте поддержку строковых литералов; см. описание ошибки в Hibernate
// под номером  HHH-8124
Root<User> u = criteria.from(User.class);
criteria.multiselect(
    u.get("username"),
    cb.selectCase()
        .when(
            cb.equal(
                cb.length(u.get("homeAddress").<String>get("zipcode")), 5
            ), "Germany"
        )
        .when(
            cb.equal(
                cb.length(u.get("homeAddress").<String>get("zipcode")), 4
            ), "Switzerland"
```

```
        )
        .otherwise("Other")
);
```

За информацией о стандартных встроенных функциях обращайтесь к таблицам в предыдущем разделе. В отличие от вызовов функций в ограничениях, встретив неизвестную функцию в проекции, Hibernate не будет отправлять ее в базу данных в виде простого вызова SQL-функции. Любая функция, которая вызывается в проекции, *должна быть* известна Hibernate и вызываться с помощью специального JPQL-оператора `function()`.

Следующая проекция вернет имя каждого аукционного товара (`Item`) и количество дней между датой его создания и датой окончания аукциона с помощью SQL-функции `datediff()` базы данных H2:

```
select
    i.name,
    function('DATEDIFF', 'DAY', i.createdOn, i.auctionEnd)
from Item i
```

```
Root<Item> i = criteria.from(Item.class);
criteria.multiselect(
    i.get("name"),
    cb.function(
        "DATEDIFF",
        Integer.class,
        cb.literal("DAY"),
        i.get("createdOn"),
        i.get("auctionEnd")
    )
);
```

Если, напротив, потребуется вызвать функцию посредственно, сначала нужно сообщить Hibernate тип ее возвращаемого значения, чтобы он мог корректно обработать запрос. Функции для использования в проекциях должны определяться в экземпляре диалекта базы данных `org.hibernate.Dialect`. Функция `datediff()`, к примеру, уже определена в диалекте базы данных H2. После этого функцию можно вызвать либо с помощью оператора `function()`, который будет работать с любыми реализациями JPA при обращении к базе H2, либо напрямую, как `datediff()`, что, скорее всего, будет работать только с Hibernate. Загляните в исходный код диалекта вашей базы данных – скорее всего, вы найдете множество других зарегистрированных там нестандартных функций SQL.

Кроме того, функции SQL можно добавлять программно, при загрузке Hibernate, вызывая метод `applySqlFunction()` интерфейса `MetadataBuilder`. В следующем примере показано, как добавить SQL-функцию `lpad()` во время загрузки Hibernate:

```
...
MetadataBuilder metadataBuilder = metadataSources.getMetadataBuilder();
metadataBuilder.applySqlFunction(
```

```
    "lpad",
    new org.hibernate.dialect.function.StandardSQLFunction(
        "lpad", org.hibernate.type.StringType.INSTANCE
    )
);
```

За более подробной информацией обращайтесь к документации Javadoc с описанием `SQLFunction` и его подклассов.

Далее мы рассмотрим агрегирующие функции, самые востребованные при создании отчетов.

15.3.5. Агрегирующие функции

Запросы, созданные для получения отчетов, часто используют встроенные возможности баз данных по группировке и агрегации данных. Например, в отчете можно показать самую большую начальную стоимость товара в каждой категории. Эти вычисления будут выполнены на уровне базы данных, и вам не нужно будет загружать много экземпляров сущностей `Item` в память.

В JPA стандартизованы следующие функции агрегирования: `count()`, `min()`, `max()`, `sum()` и `avg()`.

Следующий запрос подсчитает количество экземпляров `Item`:

```
select count(i) from Item i
```

```
criteria.select(
    cb.count(criteria.from(Item.class))
);
```

Он вернет результат в виде объекта `Long`:

```
Long count = (Long)query.getSingleResult();
```

Функция `count(distinct)` в JPQL и метод `countDistinct()` удаляют дубликаты:

```
select count(distinct i.name) from Item i
```

```
criteria.select(
    cb.countDistinct(
        criteria.from(Item.class).get("name")
    )
);
```

Следующий запрос вычислит сумму всех ставок (`Bid`):

```
select sum(b.amount) from Bid b
```

```
CriteriaQuery<Number> criteria = cb.createQuery(Number.class);
criteria.select(
    cb.sum(
        criteria.from(Bid.class).<BigDecimal>get("amount")
    )
);
```

Запрос вернет значение типа BigDecimal, поскольку свойство amount имеет тип BigDecimal. Функция sum() также распознает тип BigInteger, а для всех остальных числовых типов возвращает значение типа Long.

Следующий запрос вернет минимальную и максимальную ставки для конкретного экземпляра Item:

```
select min(b.amount), max(b.amount) from Bid b
    where b.item.id = :itemId
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.multiselect(
    cb.min(b.<BigDecimal>get("amount")),
    cb.max(b.<BigDecimal>get("amount"))
);
criteria.where(
    cb.equal(
        b.get("item").<Long>get("id"),
        cb.parameter(Long.class, "itemId")
    )
);
```

Результатом этого запроса будет кортеж объектов BigDecimal (два экземпляра BigDecimal в массиве типа Object[]).

Когда агрегирующая функция в предложении SELECT используется без группировки в предложении GROUP BY, в результате будет возвращена единственная запись с агрегированным значением. Это значит, что в отсутствие предложения GROUP BY любое предложение SELECT с агрегирующими функциями должно состоять *только* из них.

Для получения более сложных данных для отчетов нужно выполнить *группировку* данных.

15.3.6. Группировка данных

В JPA определяется поддержка нескольких особенностей SQL, часто используемых при создании отчетов (хотя они могут применяться и в других случаях). В запросах для создания отчетов сначала записывается предложение SELECT, определяющее проекцию, а затем предложения GROUP BY и HAVING, описывающие агрегирование.

Так же как в SQL, любое свойство или псевдоним, стоящие в предложении SELECT отдельно от агрегирующих функций, должны также присутствовать в предложении GROUP BY. Рассмотрим запрос, который подсчитывает количество пользователей с одинаковой фамилией:

```
select u.lastname, count(u) from User u
    group by u.lastname
```

```
Root<User> u = criteria.from(User.class);
criteria.multiselect(
```

```
    u.get("lastname"),
    cb.count(u)
);
criteria.groupBy(u.get("lastname"));
```

В данном примере поле `u.lastname` находится вне агрегирующей функции, поэтому данные в проекции должны быть сгруппированы по `u.lastname`. Также не нужно указывать свойство, которое должно подсчитываться, поскольку выражение `count(u)` будет автоматически преобразовано в `count(u.id)`.

Следующий запрос вычислит средний размер ставки (`Bid#amount`) для каждого товара `Item`:

```
select i.name, avg(b.amount)
    from Bid b join b.item i
    group by i.name
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.multiselect(
    b.get("item").get("name"),
    cb.avg(b.<BigDecimal>get("amount"))
);
criteria.groupBy(b.get("item").get("name"));
```

Особенности Hibernate

При использовании группировок можно столкнуться с некоторыми ограничениями в Hibernate. Следующий запрос полностью соответствует спецификации, но Hibernate обрабатывает его некорректно:

```
select i, avg(b.amount)
    from Bid b join b.item i
    group by i
```

Спецификация JPA разрешает указывать в группировке псевдонимы сущностей: `group by i`. Но Hibernate не подставит свойства сущности `Item` в сгенерированное предложение SQL `GROUP BY`, что вызовет несоответствие с предложением `SELECT`. Все свойства вам придется указывать вручную, пока эта ошибка не будет исправлена в Hibernate (это одна из самых старых ошибок под номером HHH-1615):

```
select i, avg(b.amount)
    from Bid b join b.item i
    group by i.id, i.name, i.createdOn, i.auctionEnd,
             i.auctionType, i.approved, i.buyNowPrice,
             i.seller
```

```
Root<Bid> b = criteria.from(Bid.class);
Join<Bid, Item> i = b.join("item");
criteria.multiselect(
    i,
    cb.avg(b.<BigDecimal>get("amount"))
```

```
);
criteria.groupBy(
    i.get("id"), i.get("name"), i.get("createdOn"), i.get("auctionEnd"),
    i.get("auctionType"), i.get("approved"), i.get("buyNowPrice"),
    i.get("seller")
);
```

Иногда требуется исключить некоторые группировки, выбирая только конкретные агрегированные значения. Ограничение для записей описывается в предложении `WHERE`. А предложение `HAVING` описывает ограничение для группировок.

Например, следующий запрос подсчитает количество пользователей с фамилией, начинающейся на букву «D»:

```
select u.lastname, count(u) from User u
    group by u.lastname
    having u.lastname like 'D%'
```

```
Root<User> u = criteria.from(User.class);
criteria.multiselect(
    u.get("lastname"),
    cb.count(u)
);
criteria.groupBy(u.get("lastname"));
criteria.having(cb.like(u.<String>get("lastname"), "D%"));
```

Предложения `SELECT` и `HAVING` подчиняются общему правилу: вне агрегирующих функций могут находиться только свойства, по которым осуществляется группировка.

В предыдущих разделах вы познакомились с основами запросов. Пришло время изучить более продвинутые возможности. У большинства инженеров наибольшие затруднения вызывает соединение произвольных данных с помощью оператора *join* – мощного механизма реляционной модели.

15.4. Соединения

Оператор *join* соединяет данные из двух (или более) отношений. Соединение данных позволяет извлечь несколько связанных экземпляров и коллекций в одном запросе: например, загрузить экземпляр `Item` и его коллекцию `bids` за одно обращение к базе данных. Сейчас мы продемонстрируем, как работают основные операции соединения и как они используются для определения стратегий *динамического извлечения*. Рассмотрим сначала, как соединения работают в обычном SQL, оставив на минуту JPA.

15.4.1. Соединения в SQL

Рассмотрим упомянутый выше пример: соединение таблиц `ITEM` (товары) и `BID` (ставки), показанных на рис. 15.1. В базе данных находятся три товара: для первого

имеются три ставки, для второго – одна, а для третьего нет ни одной. Обратите внимание, что показаны только некоторые столбцы, остальные заменены многоточиями.

ЭКЗЕМПЛЯР ITEM

ID	NAME	...
1	Foo	...
2	Bar	...
3	Baz	...

BID

ID	ITEM_ID	AMOUNT	...
1	1	99.00	...
2	1	100.00	...
3	1	101.00	...
4	2	4.99	...

Рис. 15.1 ❖ Таблицы `ITEM` и `BID` – первые кандидаты для соединения

Большинство думает об операции *join* в контексте баз данных SQL как о внутреннем соединении (*inner join*). Внутреннее соединение является наиболее важным типом соединений и самым простым для понимания. Рассмотрим выражение SQL и его результат (рис. 15.2). Это выражение SQL содержит в предложении `FROM` оператор *inner join* в стиле *ANSI*.

```
select i.*, b.*
  from ITEM i
    inner join BID b on i.ID = b.ITEM_ID
```

i.ID	i.NAME	...	b.ID	b.ITEM_ID	b.AMOUNT
1	Foo	...	1	1	99.00
1	Foo	...	2	1	100.00
1	Foo	...	3	1	101.00
2	Bar	...	4	2	4.99

Рис. 15.2 ❖ Результат внутреннего соединения двух таблиц в стиле ANSI

Выполняя внутреннее соединение таблиц `ITEM` и `BID` по условию равенства атрибута `ID` из таблицы `ITEM` атрибуту `ITEM_ID` из таблицы `BID`, вы получите все товары с их ставками. Обратите внимание, что в результат этого запроса попадут только товары, *имеющие* ставки.

Операцию соединения можно представить следующим образом: сначала берется произведение двух таблиц, т. е. все возможные комбинации записей из `ITEM` и `BID`. Затем объединенные записи фильтруются с использованием *условия соединения*: выражения в предложении `ON` (любой хороший движок базы данных использует более сложный алгоритм для создания соединения; он обычно не создает затратного произведения, чтобы затем его отфильтровать). Условие соединения – это логическое выражение, возвращающее `true`, если объединенная запись должна попасть в результат запроса.

Важно понимать, что условием соединения может быть любое выражение, возвращающее `true`. Данные можно соединять по-разному; вы не ограничены только сравнением значений идентификаторов. Например, условие соединения `on i.ID = b.ITEM_ID and b.AMOUNT > 100` будет истинным только для записей в `BID` со значением атрибута `AMOUNT` больше 100. На столбец `ITEM_ID` таблицы `BID` наложено ограничение внешнего ключа, гарантирующее присутствие в записи из `BID` ссылки на запись в `ITEM`. Но это не означает, что соединение можно выполнять, используя только столбцы первичного и внешнего ключей. Ключевые столбцы, безусловно, чаще других выступают в роли операндов в условиях соединений, поскольку часто требуется извлекать связанную информацию.

Чтобы извлечь *все* товары, а не только со ставками, нужно использовать *левое внешнее соединение* ((left) outer join), как показано на рис. 15.3.

```
select i.*, b.*
  from ITEM i
    left outer join BID b on i.ID = b.ITEM_ID
```

i.ID	i.NAME	...	b.ID	b.ITEM_ID	b.AMOUNT
1	Foo	...	1	1	99.00
1	Foo	...	2	1	100.00
1	Foo	...	3	1	101.00
2	Bar	...	4	2	4.99
3	Baz	...			

Рис. 15.3 ❖ Результат левого внешнего соединения двух таблиц в стиле ANSI

В случае левого внешнего соединения каждая запись в (левой) таблице `ITEM`, *не удовлетворяющая* условию соединения, также будет добавлена в результат, а все столбцы таблицы `BID` будут содержать для нее значения `NULL`. Правые внешние соединения применяются редко; разработчики обычно думают слева направо и в операции соединения помещают основную таблицу вначале. На рис. 15.4 показан такой же результат, но только с применением правого внешнего соединения, где ведущей выступает таблица `BID`, а не `ITEM`.

В SQL условия соединения обычно указываются явно. К сожалению, в качестве условия нельзя использовать имя ограничения внешнего ключа: `select * from ITEM join BID on FK_BID_ITEM_ID` не сработает.

Условие соединения должно быть указано либо в предложении `ON`, в соответствии с синтаксисом ANSI, либо в предложении `WHERE`, при использовании так называемого *мета-соединения*: `select * from ITEM i, BID b where i.ID = b.ITEM_ID`. Это пример внутреннего соединения; как видите, в предложении `FROM` сначала создается произведение таблиц.

Теперь пришло время изучить возможности JPA. Не забывайте, что в конечном итоге Hibernate превращает все запросы в их SQL-эквиваленты, поэтому даже

если синтаксис будет слегка отличаться, вы всегда сможете вернуться к примерам из этого раздела, чтобы проверить свое понимание итогового кода SQL и результатов запросов.

```
select b.*, i.*
  from BID b
    right outer join ITEM i on b.ITEM_ID = i.ID
```

b.ID	b.ITEM_ID	b.AMOUNT	i.ID	i.NAME	...
1	1	99.00	1	Foo	...
2	1	100.00	1	Foo	...
3	1	101.00	1	Foo	...
4	2	4.99	2	Bar	...
			3	Baz	...

Рис. 15.4 ❖ Результат правого внешнего соединения двух таблиц в стиле ANSI

15.4.2. Соединение таблиц в JPA

JPA поддерживает четыре способа описания соединений (внешних и внутренних) в запросе:

- *неявное* соединение по связи с использованием выражений для представления путей к атрибутам;
- *обычное* соединение в предложении `FROM` с помощью оператора `join`;
- *немедленное* соединение в предложении `FROM` с использованием оператора `join` и ключевого слова `fetch` для немедленного извлечения;
- *тета*-соединение в предложении `WHERE`.

Рассмотрим сначала неявные соединения по связи.

15.4.3. Неявные соединения по связи

В запросах JPA необязательно явно указывать условия соединения. Достаточно указать имя связи, отображаемой в классе Java. Как этого недостает в SQL – возможности описывать условие соединения с помощью ограничения внешнего ключа. Поскольку большинство отношений по внешнему ключу описывается в схеме базы данных, имена этих отображаемых связей можно использовать в языке запросов. Пусть это всего лишь синтаксический сахар, но он очень удобен.

Например, класс сущности `Bid` имеет отображаемую связь *многие к одному* с классом `Item` (с именем `item`). Если указать эту связь в запросе, Hibernate получит достаточно информации, чтобы сформировать условие соединения со сравнением ключевых столбцов. Благодаря этому запросы становятся короче и понятнее.

Ранее в этой главе вы уже сталкивались с выражениями, описывающими пути к атрибутам с помощью точечной нотации: выражения, ссылающиеся на единственное значение, такие как `user.homeAddress.zipcode`, и выражения, ссылающие-

ся на коллекцию, такие как `item.bids`. Такие выражения можно использовать в запросах с неявными соединениями:

```
select b from Bid b where b.item.name like 'Fo%'
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.like(
        b.get("item").<String>get("name"),
        "Fo%"
    )
);
```

Путь `b.item.name` создаст неявное соединение по связи *многие к одному* от сущности `Bid` к сущности `Item`; эта связь называется `item`. Hibernate знает, что вы отобразили эту связь с помощью внешнего ключа `ITEM_ID` в таблице `BID`, и сможет правильно сформировать условие соединения в SQL. Неявные соединения всегда работают для ассоциаций *многие к одному* и *один ко многим*, но не для ассоциаций с коллекциями (нельзя написать `item.bids.amount`).

Одному выражению пути может соответствовать несколько соединений:

```
select b from Bid b where b.item.seller.username = 'johndoe'
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.equal(
        b.get("item").get("seller").get("username"),
        "johndoe"
    )
);
```

Этот запрос соединяет таблицы `BID`, `ITEM` и `USER`.

Мы не советуем применять этот стиль в более сложных запросах. Соединения играют важную роль в SQL, поэтому во время оптимизации запросов очень важно иметь возможность с ходу определять количество соединений. Рассмотрим следующий запрос:

```
select b from Bid b where b.item.seller.username = 'johndoe'
    and b.item.buyNowPrice is not null
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.and(
        cb.equal(
            b.get("item").get("seller").get("username"),
            "johndoe"
        ),
        cb.isNotNull(b.get("item").get("buyNowPrice"))
    )
);
```

Сколько соединений потребуется, чтобы выразить этот запрос на языке SQL? Даже если вы правильно ответите на вопрос, это отнимет у вас несколько секунд. Правильный ответ – два. Сформированный код SQL будет выглядеть примерно так:

```
select b.*
    from BID b
        inner join ITEM i on b.ITEM_ID = i.ID
        inner join USER u on i.SELLER_ID = u.ID
        where u.USERNAME = 'johndoe'
            and i.BUYNOWPRICE is not null;
```

Альтернативой соединениям с такими сложными выражениями для описания путей служат обычные соединения с предложениями `FROM`.

15.4.4. Явные соединения

JPA различает цели соединений. Предположим, что вы выбираете все товары; соединение с таблицей ставок может преследовать две цели.

Вы можете ограничить количество товаров, возвращаемое запросом, на основе какого-либо критерия, применяемого к ставкам. Например, можно получить все товары со ставками больше 100, что потребует *внутреннего соединения*. В этом случае вас не будут интересовать товары без ставок.

С другой стороны, главной целью может быть получение всех товаров, а ставки присоединяются только затем, чтобы получить все данные в одном выражении SQL, что мы ранее назвали *немедленным извлечением через соединение*. Помните, что по умолчанию мы отображаем все связи как *отложенные*, поэтому немедленное извлечение переопределит стратегию извлечения по умолчанию в конкретном запросе во время выполнения.

Давайте сначала напишем несколько запросов, использовав соединения для ограничения результатов. Чтобы извлечь все экземпляры `Item`, оставив только со ставками, превышающими определенное значение, нужно присвоить псевдоним связи соединения. Затем на него можно будет сослаться в предложении `WHERE` для описания ограничения:

```
select i from Item i
    join i.bids b
    where b.amount > 100
```

```
Root<Item> i = criteria.from(Item.class);
Join<Item, Bid> b = i.join("bids");
criteria.select(i).where(
    cb.gt(b.<BigDecimal>get("amount"), new BigDecimal(100))
);
```

В этом запросе коллекции `bids` присвоен псевдоним `b`, с помощью которого описывается ограничение для экземпляров `Item`, чтобы извлечь только те, которые имеют значение `Bid#amount` больше 100.

До сих пор в этом разделе вы видели только внутренние соединения. Внешние соединения в основном применяются для динамического извлечения, которое мы

скоро обсудим. Но иногда бывает нужно написать простой запрос с внешним соединением без применения динамической стратегии извлечения. Следующий запрос, к примеру, извлечет товары без ставок и со ставками больше минимального значения:

```
select i, b from Item i
    left join i.bids b on b.amount > 100
```

```
Root<Item> i = criteria.from(Item.class);
Join<Item, Bid> b = i.join("bids", JoinType.LEFT);
b.on(
    cb.gt(b.<BigDecimal>get("amount"), new BigDecimal(100))
);
criteria.multiselect(i, b);
```

Запрос вернет кортеж объектов `Item` и `Bid` в виде коллекции `List<Object[]>`.

Первое, что бросается в глаза, – ключевое слово `LEFT` и аргумент `JoinType.LEFT` в запросе на основе критериев. Вы можете использовать форму `LEFT OUTER JOIN` и `RIGHT OUTER JOIN` в запросах JPQL, но мы предпочитаем короткий вариант.

Вторая особенность – дополнительное условие соединения после ключевого слова `ON`. Если поместить в предложение `WHERE` условие `b.amount > 100`, в результат запроса попадут только экземпляры `Item`, имеющие ставки. Но это не то, что нам надо: нам нужно извлечь товары и ставки, а также товары без ставок. Если товар *имеет* ставку, она должна быть больше 100. Добавляя дополнительное условие соединения в предложение `FROM`, мы накладываем ограничение на экземпляры `Bid`, по-прежнему возвращая все экземпляры `Item`, независимо от наличия ставок.

Дополнительное условие соединения будет преобразовано в код SQL:

```
... from ITEM i
    left outer join BID b
        on i.ID = b.ITEM_ID and (b.AMOUNT > 100)
```

Запрос SQL всегда будет содержать неявное условие соединения отображаемой связи `i.ID = b.ITEM_ID`. В условие соединения можно добавить лишь дополнительное выражение. JPA и Hibernate не поддерживают произвольных внешних соединений без применения отображаемых связей сущностей или коллекций.

В Hibernate есть нестандартное ключевое слово `WITH` – эквивалент `ON` в JPQL. Его можно встретить в старом коде, поскольку поддержка `ON` в JPA была стандартизована лишь недавно.

Можно написать запрос, возвращающий те же данные, но с помощью правого внешнего соединения, поменяв ведущую таблицу:

```
select b, i from Bid b
    right outer join b.item i
    where b is null or b.amount > 100
```

```
Root<Bid> b = criteria.from(Bid.class);
Join<Bid, Item> i = b.join("item", JoinType.RIGHT);
criteria.multiselect(b, i).where(
```

```
    cb.or(
        cb.isNull(b),
        cb.gt(b.<BigDecimal>get("amount"), new BigDecimal(100)))
);
```

Правое внешнее соединение гораздо важнее, чем вы могли бы подумать. Ранее в этой книге мы советовали избегать по мере возможности отображения хранимых коллекций. Поэтому если у вас нет коллекции типа *один ко многим* `Item#bids`, вам понадобится правое внешнее соединение, чтобы извлечь все экземпляры `Item` с соответствующими им экземплярами `Bid`. Запрос начинается с «другой» стороны – отображения *многие к одному* `Bid#item`.

Отложенные внешние соединения также играют важную роль в немедленном динамическом извлечении.

15.4.5. Динамическое извлечение с помощью соединений

Все запросы из предыдущих разделов имели кое-что общее: все возвращаемые экземпляры `Item` имели коллекцию `bids`. Когда коллекция, отмеченная аннотацией `@OneToMany`, отображается с параметром `FetchType.LAZY` (по умолчанию для коллекций), она будет инициализирована с помощью отдельного выражения SQL при первом обращении к ней. Так же работают все отношения, связанные только с одной сущностью, такие как связь `seller` с аннотацией `@ManyToOne` у каждого экземпляра `Item`. По умолчанию Hibernate создаст прокси-объект и выполнит отложенную загрузку связанного экземпляра `User` только при первом обращении.

Но что делать, если нужно изменить это поведение? Во-первых, можно поменять план извлечения в метаданных и отобразить коллекцию или связь с параметром `FetchType.EAGER`. После этого Hibernate выполнит весь необходимый код SQL, чтобы обеспечить загрузку требуемого графа объектов. Это, в свою очередь, означает, что один запрос в JPA может привести к выполнению нескольких операций в SQL! Например, простой запрос `select i from Item i` может привести к выполнению дополнительных выражений SQL для загрузки коллекции `bids` каждого объекта `Item`, свойства `seller` каждого объекта `Item` и т. д.

В главе 12 мы показали пример с глобальным планом отложенного извлечения, который определялся в метаданных отображения, где мы не должны были использовать параметра `FetchType.EAGER` для коллекций и связей. Но затем, для конкретного варианта использования, *динамически* переопределили план отложенного извлечения и написали запрос, извлекающий данные самым эффективным способом. Например, не нужно выполнять несколько выражений SQL, чтобы извлечь все экземпляры `Item` с инициализированными коллекциями `bids` и установленным значением `seller` каждого экземпляра `Item`. Это можно сделать одним выражением `SQL` с помощью операции соединения.

Немедленное извлечение связанных данных осуществляется с помощью ключевого слова `FETCH` в JPQL и с помощью метода `fetch()` в API запросов на основе критериев.

```
select i from Item i
    left join fetch i.bids
```

```
Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
criteria.select(i);
```

Вы уже видели код SQL, в который транслируется этот запрос, а также результат запроса на рис. 15.3.

Запрос возвращает список `List<Item>`; каждый экземпляр `Item` вместе со своей коллекцией `bids` полностью инициализирован. Это отличается от тех кортежей, что вы получали в предыдущем разделе!

Но будьте внимательны – возможно, вы не ожидаете получить от предыдущего запроса повторяющихся результатов:

```
List<Item> result = query.getResultList();
assertEquals(result.size(), 5);          ← 3 товара, 4 ставки, 5 записей в результате

Set<Item> distinctResult = new LinkedHashSet<Item>(result);  ← Всего лишь три товара
assertEquals(distinctResult.size(), 3);  ← Удаление дубликатов в памяти
```

Убедитесь, что понимаете, почему повторяющиеся записи появились в результирующей коллекции `List`. Проверьте еще раз количество «записей» `Item` в результате, как показано на рис. 15.3. Hibernate превратит записи в элементы списка; но вам может понадобиться точное количество записей для вывода таблицы в отчете.

Вы можете избавиться от повторяющихся экземпляров `Item`, передав получившийся список `List` в конструктор множества `LinkedHashSet`, которое удалит дубликаты, но сохранит порядок элементов. Также Hibernate может удалять повторяющиеся элементы с помощью операции `DISTINCT` и метода `distinct()` в запросе на основе критериев:

```
select distinct i from Item i
    left join fetch i.bids
```

```
Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
criteria.select(i).distinct(true);
```

Обратите внимание, что в данном случае операция `DISTINCT` выполнится *не* в базе данных. В запросе SQL не будет ключевого слова `DISTINCT`. Фактически вы *не можете* удалить повторяющихся записей на уровне объекта `ResultSet` в SQL. Hibernate удалит их в памяти так же, как вы делали это с помощью `LinkedHashSet`.

Такой же синтаксис может использоваться для немедленного извлечения связей *многие к одному* и *один к одному*:

```
select distinct i from Item i
    left join fetch i.bids b
        join fetch b.bidder
        left join fetch i.seller
```

```
Root<Item> i = criteria.from(Item.class);
```

```
Fetch<Item, Bid> b = i.fetch("bids", JoinType.LEFT);
b.fetch("bidder");
i.fetch("seller", JoinType.LEFT);
criteria.select(i).distinct(true);
```

Столбцы внешних ключей не могут содержать null. Внутреннее соединение или внешнее – в данном случае не важно

Этот запрос вернет список `List<Item>`, и каждый экземпляр `Item` получит инициализированную коллекцию `bids`. Свойство `seller` каждого экземпляра `Item` также будет загружено. И наконец, будет инициализировано свойство `bidder` (пользователь, сделавший ставку) каждого экземпляра `Bid`. Это можно сделать в одном запросе SQL, соединив таблицы `ITEM`, `BID` и `USERS`.

Если написать `JOIN FETCH` без ключевого слова `LEFT`, будет выполнена отложенная загрузка с внутренним соединением (как если бы вы написали `INNER JOIN FETCH`). Немедленное внутреннее соединение логично использовать, когда извлекаемые объекты гарантированно существуют: экземпляр `Item` должен иметь установленное свойство `seller`, а экземпляр `Bid` – свойство `bidder`.

Количество одновременно извлекаемых связей при немедленной загрузке, как и число возвращаемых запросом записей, ограничено. Рассмотрим следующий запрос, который инициализирует коллекции `Item#bids` и `Item#images`:

```
select distinct i from Item i
    left join fetch i.bids
    left join fetch i.images
```

```
Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
i.fetch("images", JoinType.LEFT);
criteria.select(i).distinct(true);
```

← Декартово произведение – это плохо

Этот запрос плох тем, что создает декартово произведение коллекций `bids` и `images` и может вернуть результат огромного объема. Мы рассмотрели эту проблему в разделе 12.2.2.

Стратегия немедленного динамического извлечения в запросах имеет следующие подводные камни.

- Никогда не присваивайте псевдонимы любым связям, извлекаемым немедленно, для использования в ограничениях или проекциях. Запрос `left join fetch i.bids b where b.amount ...` не сработает. Нельзя сказать Hibernate: «Загрузи экземпляры `Item` и инициализируй коллекцию `bids` только экземплярами `Bid` с определенным значением». Вы *можете* присваивать псевдонимы немедленно извлекаемым связям для дальнейшего уточнения, например для извлечения поля `bidder` каждого объекта `Bid`: `left join fetch i.bids b join fetch b.bidder`.
- Не извлекайте более одной коллекции; в противном случае вы получите декартово произведение. Но связей с единственным значением можно извлечь любое количество – это не приведет к созданию произведения.
- Запросы игнорируют любые стратегии извлечения, определяемые в метаданных отображения с помощью аннотации `@org.hibernate.annotations.`

Fetch. Например, отображение коллекции bids с параметром org.hibernate.annotations.FetchMode.JOIN никак не повлияет на выполняемые запросы. Динамическая стратегия извлечения игнорирует глобальную стратегию извлечения. С другой стороны, Hibernate никогда не игнорирует отображаемого *плана извлечения*: Hibernate всегда выбирает параметр FetchType.EAGER, и во время выполнения запроса вы можете увидеть несколько дополнительных выражений SQL.

- При немедленном извлечении коллекции возвращаемый Hibernate список List содержит столько же записей, сколько вернул запрос SQL, в том числе повторяющиеся ссылки. Вы можете избавиться от дубликатов в памяти – вручную, с помощью LinkedHashSet, или с помощью ключевого слова DISTINCT в запросе.

Есть еще один нюанс, заслуживающий внимания. При немедленном извлечении коллекции вы не сможете извлекать результаты запроса из базы данных постранично. Как, например, должен отреагировать запрос select i from Item i fetch i.bids на параметры Query#setFirstResult(21) и Query#setMaxResults(10)?

Очевидно, что вы ожидаете извлечь 10 товаров, начиная с 21-го. Но при этом вы хотите немедленно извлечь всю коллекцию bids экземпляра Item. В таком случае база данных не сможет обеспечить постраничного вывода; вы не сможете ограничить результат запроса SQL десятью произвольными записями. Если в запросе коллекция извлекается немедленно, Hibernate выполнит постраничную выборку в памяти приложения. Это означает, что в памяти окажутся *все* экземпляры Item, и каждый будет иметь инициализированную коллекцию bids. После этого Hibernate вернет требуемую страницу результатов, например только товары с 21 по 30.

Но в памяти могут поместиться не все товары, и вы, написав такой запрос, вероятно, ожидали, что постраничная выборка будет выполняться на стороне базы данных! Поэтому Hibernate запишет в журнал предупреждение, если встретит в запросе fetch [collectionPath] и вы вызвали setFirstResult() или setMaxResults().

Мы не рекомендуем использовать fetch [collectionPath] вместе с вызовами setMaxResults() и setFirstResult(). Как правило, всегда можно написать более простой запрос, извлекающий данные, необходимые для отображения, – мы надеемся, что вы не станете использовать постраничную выборку для изменения данных. Если вам, к примеру, потребуется показать несколько страниц с товарами и напротив каждого указать количество сделанных ставок, используйте следующий запрос:

```
select i.id, i.name, count(b)
    from Item i left join i.bids b
        group by i.id, i.name
```

Результат этого запроса база данных сможет вывести постранично после вызова методов setFirstResult() и setMaxResults(). Это будет гораздо эффективнее, чем извлечение *любых* экземпляров Item или Bid в память; пусть всю работу сделает база данных.

Последним способом соединения в JPA является *мета-соединение*.

15.4.6. Тета-соединения

В традиционном SQL тета-соединение – это декартово произведение с условием соединения в предложении `WHERE`, ограничивающим это произведение. В запросах JPA синтаксис тета-соединения применяется, когда условие соединения не является связью по внешнему ключу, отображаемой на связь класса.

Предположим, что вы используете имя пользователя (`User`) для записи в журнал, а не отображаете связь между `LogRecord` и `User`. Этим классам ничего не известно друг о друге, поскольку они не связаны. Вы можете найти все экземпляры `User` и соответствующие им записи в таблице `LogRecord` с помощью следующего тета-соединения:

```
select u, log from User u, LogRecord log
    where u.username = log.username
```

```
Root<User> u = criteria.from(User.class);
Root<LogRecord> log = criteria.from(LogRecord.class);
criteria.where(
    cb.equal(u.get("username"), log.get("username")));
criteria.multiselect(u, log);
```

Условие соединения в данном случае сравнивает атрибуты `username` обоих классов. Если обе записи будут иметь одинаковое значение `username`, они попадут в результат. Результат запроса будет состоять из кортежей:

```
List<Object[]> result = query.getResultList();
for (Object[] row : result) {
    assertTrue(row[0] instanceof User);
    assertTrue(row[1] instanceof LogRecord);
}
```

Возможно, вы будете прибегать к тета-соединениям не слишком часто. Обратите внимание, что в настоящий момент в JPA нельзя задать внешнее соединение для таблиц, не связанных отображаемой связью; тета-соединения – это внутренние соединения.

Другим распространенным случаем применения тета-соединений является сравнение первичного или внешнего ключа с параметрами запроса или с другими внешними ключами в предложении `WHERE`:

```
select i, b from Item i, Bid b
    where b.item = i and i.seller = b.bidder
```

```
Root<Item> i = criteria.from(Item.class);
Root<Bid> b = criteria.from(Bid.class);
criteria.where(
    cb.equal(b.get("item"), i),
    cb.equal(i.get("seller"), b.get("bidder"))
);
criteria.multiselect(i, b);
```

Этот запрос вернет пары экземпляров `Item` (товар) и `Bid` (ставка), где пользователь, сделавший ставку (`bidder`), является также продавцом (`seller`). Для CaveatEmptor этот запрос играет важную роль, поскольку позволяет вычислить пользователей, делающих ставки за свой собственный товар. Возможно, этот запрос стоит преобразовать в ограничение базы данных, чтобы не допустить сохранения подобных экземпляров `Bid`.

В предыдущем запросе также присутствует интересное выражение сравнения: `i.seller = b.bidder`. Это сравнение идентификаторов, которое рассматривается в следующем разделе.

15.4.7. Сравнение идентификаторов

JPA поддерживает синтаксис неявного сравнения идентификаторов в запросах:

```
select i, u from Item i, User u
    where i.seller = u and u.username like 'j%'
```

```
Root<Item> i = criteria.from(Item.class);
Root<User> u = criteria.from(User.class);
criteria.where(
    cb.equal(i.get("seller"), u),
    cb.like(u.<String>get("username"), "j%")
);
criteria.multiselect(i, u);
```

В этом запросе свойство `i.seller` ссылается на столбец внешнего ключа `SELLER_ID` таблицы `ITEM`, который ссылается на таблицу `USERS`. Псевдоним `u` ссылается на первичный ключ таблицы `USERS` (столбец `ID`). Следовательно, этот запрос с тета-соединением является более простым эквивалентом следующего:

```
select i, u from Item i, User u
    where i.seller.id = u.id and u.username like 'j%'
```

```
Root<Item> i = criteria.from(Item.class);
Root<User> u = criteria.from(User.class);
criteria.where(
    cb.equal(i.get("seller").get("id"), u.get("id")),
    cb.like(u.<String>get("username"), "j%")
);
criteria.multiselect(i, u);
```

Особенности Hibernate

Выражение пути к атрибуту, оканчивающееся на `id`, имеет особое значение в Hibernate: имя `id` всегда ссылается на свойство идентификатора сущности. Не важно, какое имя на самом деле имеет свойство идентификатора, отмеченное аннотацией `@Id`; к нему всегда можно обратиться как `псевдонимСущности.id`. Поэтому мы советуем всегда давать свойству идентификатора имя `id`, чтобы избежать путаницы в за-

просах. Обратите внимание, что это не является требованием JPA; особое значение имени `id` придается только в Hibernate.

Также может понадобиться сравнить свойство с параметром запроса, например чтобы найти товары (`Item`) конкретного продавца (`User`):

```
select i from Item i where i.seller = :seller
```

```
Root<Item> i = criteria.from(Item.class);
criteria.where(
    cb.equal(
        i.get("seller"),
        cb.parameter(User.class, "seller")
    )
);
criteria.select(i);
query.setParameter("seller", someUser);
List<Item> result = query.getResultList();
```

Этот запрос также можно выразить в терминах идентификаторов, а не ссылок на объекты. Следующие запросы эквивалентны предыдущим:

```
select i from Item i where i.seller.id = :sellerId
```

```
Root<Item> i = criteria.from(Item.class);
criteria.where(
    cb.equal(
        i.get("seller").get("id"),
        cb.parameter(Long.class, "sellerId")
    )
);
criteria.select(i);
query.setParameter("sellerId", USER_ID);
List<Item> result = query.getResultList();
```

Что касается идентификаторов, есть существенная разница межу этой парой запросов

```
select b from Bid b where b.item.name like 'Fo%'
```

```
Root<Bid> b = criteria.from(Bid.class);
criteria.select(b).where(
    cb.like(
        b.get("item").<String>get("name"),
        "Fo%"
    )
);
```

и этой:

```
select b from Bid b where b.item.id = :itemId
```

```
CriteriaQuery<Bid> criteria = cb.createQuery(Bid.class);
Root<Bid> b = criteria.from(Bid.class);
```

```
criteria.where(
    cb.equal(
        b.get("item").get("id"),
        cb.parameter(Long.class, "itemId")
    )
);
criteria.select(b);
```

В первой паре используется неявное соединение таблиц; во второй – соединений нет вообще!

На этом мы заканчиваем обсуждение запросов с соединениями. Нашей последней темой станут запросы внутри запросов, т. е. *подзапросы*.

15.5. Подзапросы

Подзапросы – это важная и мощная возможность SQL. Подзапрос – это запрос, встроенный в другой запрос, как правило, в предложении `SELECT`, `FROM` или `WHERE`.

JPA допускает применение подзапросов только в предложении `WHERE`. Подзапросы в предложении `FROM` не поддерживаются, поскольку в языках запросов отсутствует *транзитивное замыкание*. Результат запроса может быть непригоден для дальнейшей выборки в предложении `FROM`. Подзапросы в предложении `SELECT` также не поддерживаются, но можно отображать подзапросы на вычисляемые свойства с помощью аннотации `@org.hibernate.annotations.Formula`, как показано в разделе 5.1.3.

Подзапросы также могут быть коррелированными.

15.5.1. Коррелированные и некореллированные подзапросы

Подзапрос может вернуть одну или несколько записей. Как правило, подзапросы, возвращающие одну запись, выполняют агрегацию. Следующий подзапрос вернет количество товаров, проданных пользователем; внешний запрос вернет всех пользователей, продавших более одного товара:

```
select u from User u
    where (
        select count(i) from Item i where i.seller = u
    ) > 1
```

```
Root<User> u = criteria.from(User.class);

Subquery<Long> sq = criteria.subquery(Long.class);
Root<Item> i = sq.from(Item.class);
sq.select(cb.count(i))
    .where(cb.equal(i.get("seller"), u)
    );

criteria.select(u);
criteria.where(cb.greaterThan(sq, 1L));
```

Внутренний запрос является *коррелированным* – он обращается к псевдониму u из внешнего запроса.

В следующем примере используется *некоррелированный подзапрос*:

```
select b from Bid b
    where b.amount + 1 >= (
        select max(b2.amount) from Bid b2
    )
```

```
Root<Bid> b = criteria.from(Bid.class);

Subquery<BigDecimal> sq = criteria.subquery(BigDecimal.class);
Root<Bid> b2 = sq.from(Bid.class);
sq.select(cb.max(b2.<BigDecimal>get("amount")));
criteria.select(b);
criteria.where(
    cb.greaterThanOrEqualTo(
        cb.sum(b.<BigDecimal>get("amount"), new BigDecimal(1)),
        sq
    )
);
```

Здесь подзапрос возвращает самую большую ставку во всем приложении; внешний запрос найдет все ставки со значениями, отличающимися от наибольшей не более чем на единицу (долларов, евро и т. д.). Обратите внимание, что в обоих примерах подзапрос в JPQL окружают скобки. Это обязательное требование.

Некоррелированные подзапросы безвредны, поэтому используйте их, когда это удобно. Такие запросы всегда можно разделить на два отдельных запроса, поскольку они не ссылаются друг на друга. Но не забывайте оценивать производительность коррелированных подзапросов. В больших базах данных стоимость простого коррелированного подзапроса сравнима со стоимостью соединения. Но переписать коррелированный подзапрос в виде двух отдельных запросов возможно не всегда.

В случае, когда подзапрос возвращает несколько записей, к нему применяются *кванторы*.

15.5.2. Кванторы

В стандарте определены следующие кванторы:

- `ALL` – выражение вернет `true`, если результат сравнения будет истинным для всех значений в результатах подзапроса. Если хотя бы для одного значения условие не выполнится, выражение вернет `false`;
- `ANY` – выражение вернет `true`, если результат сравнения будет истинным хотя бы для одного (любого) значения в результате подзапроса. Если подзапрос не вернет результатов или ни одно значение не удовлетворяет условию сравнения, выражение вернет `false`. Ключевое слово `SOME` является синонимом для `ANY`;
- `EXISTS` – выражение вернет `true`, если подзапрос вернет одно или более значений.

Например, следующий запрос найдет товары со ставками не выше 10:

```
select i from Item i
    where 10 >= all (
        select b.amount from i.bids b
    )
```

```
Root<Item> i = criteria.from(Item.class);

Subquery<BigDecimal> sq = criteria.subquery(BigDecimal.class);
Root<Bid> b = sq.from(Bid.class);
sq.select(b.<BigDecimal>get("amount"));
sq.where(cb.equal(b.get("item"), i));

criteria.select(i);
criteria.where(
    cb.greaterThanOrEqualTo(
        cb.literal(new BigDecimal(10)),
        cb.all(sq)
    )
);
```

Следующий запрос вернет товары со ставками, равными 101:

```
select i from Item i
    where 101.00 = any (
        select b.amount from i.bids b
    )
```

```
Root<Item> i = criteria.from(Item.class);

Subquery<BigDecimal> sq = criteria.subquery(BigDecimal.class);
Root<Bid> b = sq.from(Bid.class);
sq.select(b.<BigDecimal>get("amount"));
sq.where(cb.equal(b.get("item"), i));

criteria.select(i);
criteria.where(
    cb.equal(
        cb.literal(new BigDecimal("101.00")),
        cb.any(sq)
    )
);
```

Чтобы найти все товары со ставками, примените к результату подзапроса квантор `EXISTS`:

```
select i from Item i
    where exists (
        select b from Bid b where b.item = i
    )
```

```
Root<Item> i = criteria.from(Item.class);

Subquery<Bid> sq = criteria.subquery(Bid.class);
```

```
Root<Bid> b = sq.from(Bid.class);
sq.select(b).where(cb.equal(b.get("item"), i));

criteria.select(i);
criteria.where(cb.exists(sq));
```

Этот запрос гораздо важнее, чем кажется. Найти все товары со ставками можно также с помощью запроса: `select i from Item i where i.bids is not empty`. Но это требует наличия отображаемой коллекции типа *один ко многим* `Item#bids`. Если вы следуете нашим рекомендациям, вы, скорее всего, отобразили «обратную» сторону отношения: связь типа *многие к одному* `Bid#item`. Но тот же результат можно получить с помощью квантора `exists()` и подзапроса.

Подзапросы – это продвинутая технология; вас всегда должно настораживать большое количество подзапросов, поскольку запросы с ними часто можно переписать с использованием соединений и функций. Но иногда и они могут быть полезными.

15.6. Резюме

- Если до прочтения этой главы вы уже были знакомы с SQL, теперь вы сможете писать различные запросы с помощью JPQL или API запросов на основе критериев. Если вы чувствуете себя неуверенно с SQL, обращайтесь к справочному разделу.
- С помощью выборки описывается источник(и) данных – «таблицы», для которых пишется запрос. Затем применяются критерии ограничения, чтобы получить из источника требуемое подмножество «записей». Проекция определяет, какие «столбцы» будут возвращены запросом. Также есть возможность эффективно агрегировать данные на уровне базы данных.
- Мы рассмотрели соединения: как выбирать, ограничивать и объединять данные из нескольких таблиц. Приложению, использующему JPA, соединения нужны для немедленной загрузки экземпляров сущностей и коллекций за одно обращение к базе данных. Это особенно важно, когда требуется уменьшить нагрузку на базу данных, и мы советуем вам еще раз просмотреть все примеры, чтобы точно понимать работу соединений и стратегии немедленного извлечения данных.
- Запросы можно вкладывать в другие запросы, создавая подзапросы.

Глава 16

Дополнительные возможности запросов

В этой главе:
- преобразование результатов запросов;
- фильтрация коллекций;
- создание запросов на основе критериев с помощью Hibernate.

В этой главе рассматриваются дополнительные возможности запросов: преобразование результатов, фильтрация коллекций и средства в Hibernate для создания запросов на основе критериев. Сначала рассмотрим интерфейс `ResultTransformer`, который позволяет применять преобразования, отличные от используемых в Hibernate по умолчанию.

В предыдущих главах мы советовали быть осторожнее при отображении коллекций, поскольку это овчинка редко стоит выделки. В этой главе мы познакомим вас с *фильтрами для коллекций* – оригинальной функциональностью Hibernate, позволяющей более эффективно использовать хранимые коллекции. Наконец, мы познакомим вас с нестандартным интерфейсом `org.hibernate.Criteria`, а также рассмотрим ситуации, когда его лучше использовать вместо стандартных запросов на основе критериев JPA.

Начнем с преобразования результатов запросов.

Особенности Hibernate

16.1. Преобразование результатов запросов

С помощью специального преобразователя можно отфильтровать результат запроса или обработать его с помощью своей процедуры. В Hibernate определено несколько преобразователей, которые можно замещать или настраивать.

Мы предполагаем преобразовать результат обычного запроса, но для этого нам понадобится получить доступ к оригинальному интерфейсу `org.hibernate.Query` с помощью экземпляра `Session`, как показано в листинге 16.1.

Листинг 16.1 ❖ Простой запрос с проекцией нескольких атрибутов

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
Session session = em.unwrap(Session.class);
org.hibernate.Query query = session.createQuery(
    "select i.id as itemId, i.name as name, i.auctionEnd as auctionEnd from
    Item i"
);
```

Без каких-либо преобразований этот запрос вернет список `List` с элементами типа `Object[]`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
List<Object[]> result = query.list();

for (Object[] tuple : result) {
    Long itemId = (Long) tuple[0];
    String name = (String) tuple[1];
    Date auctionEnd = (Date) tuple[2];
    // ...
}
```

Каждый массив объектов – это «запись» из результатов запроса. К элементу кортежа можно обратиться по индексу: 0 – соответствует атрибуту типа `Long`, 1 - `String`, а 2 – `Date`.

Первый преобразователь, который мы рассмотрим, позволяет получить список `List` с элементами типа `List`.

Преобразование результатов запроса на основе критериев

Все примеры в этом разделе написаны для запросов JPQL, созданных с помощью `org.hibernate.Query`. Если создать объект запроса JPA типа `CriteriaQuery` с помощью интерфейса `CriteriaBuilder`, вы не сможете применить к нему преобразователь `org.hibernate.transform.ResultTransformer`: этот интерфейс доступен только в Hibernate. Даже если вы получите оригинальный Hibernate API для своего запроса на основе критериев (путем приведения к типу `HibernateQuery`, как показано в разделе 14.1.3), вы все равно не сможете применить произвольного преобразователя. К объектам запросов JPA типа `CriteriaQuery` Hibernate применяет встроенный преобразователь, соответствующий спецификации JPA; применение произвольного преобразователя переопределит это поведение и создаст проблемы. Однако для JPQL-запросов, созданных с помощью `javax.persistence.Query`, можно установить свой преобразователь, получив доступ к оригинальному интерфейсу `HibernateQuery`. Кроме того, далее вы увидите оригинальный интерфейс `org.hibernate.Criteria` – альтернативный механизм для запросов на основе критериев, поддерживающий возможность переопределения `org.hibernate.transform.ResultTransformer`.

16.1.1. Получение списка списков

Предположим, что вы хотите получить доступ по индексу, но вам не нравится тип `Object[]`. Вместо списка элементов `Object[]` каждый кортеж можно представить как список `List`, задействовав преобразователь `ToListResultTransformer`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
query.setResultTransformer(
    ToListResultTransformer.INSTANCE
);

List<List> result = query.list();

for (List list : result) {
    Long itemId = (Long) list.get(0);
    String name = (String) list.get(1);
    Date auctionEnd = (Date) list.get(2);
    // ...
}
```

Отличие совсем незначительное, но это может быть удобно, если другие уровни приложения уже работают со списками списков.

Следующий преобразователь представляет каждый кортеж в виде словаря `Map`, отображающего псевдонимы в соответствующие элементы проекции.

16.1.2. Получение списка словарей

Преобразователь `AliasToEntityMapResultTransformer` возвращает список элементов типа `java.util.Map`: по одному на каждую «запись». Псевдонимами в запросе являются `itemId`, `name` и `auctionEnd`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
query.setResultTransformer(
    AliasToEntityMapResultTransformer.INSTANCE
);

List<Map> result = query.list();

assertEquals(  ← Псевдонимы, используемые в запросе
    query.getReturnAliases(),
    new String[]{"itemId", "name", "auctionEnd"}
);

for (Map map : result) {
    Long itemId = (Long) map.get("itemId");
    String name = (String) map.get("name");
    Date auctionEnd = (Date) map.get("auctionEnd");
    // ...
}
```

Если псевдонимы в запросе неизвестны, их можно получить динамически, вызвав метод `org.hibernate.Query#getReturnAliases()`.

В нашем примере запрос возвращает скалярные значения, но вам также может понадобиться преобразовывать результаты, содержащие экземпляры сущностей. Следующий пример использует псевдонимы для сущностей из проекции и список `List` элементов типа `Map`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
org.hibernate.Query entityQuery = session.createQuery(
    "select i as item, u as seller from Item i join i.seller u"
);

entityQuery.setResultTransformer(
    AliasToEntityMapResultTransformer.INSTANCE
);

List<Map> result = entityQuery.list();

for (Map map : result) {
    Item item = (Item) map.get("item");
    User seller = (User) map.get("seller");

    assertEquals(item.getSeller(), seller);
    // ...
}
```

Еще большую пользу может принести следующий преобразователь, отображающий атрибуты результатов запроса в свойства компонента JavaBean по их псевдонимам.

16.1.3. Отображение атрибутов в свойства компонента JavaBean

В разделе 15.3.2 мы показали, как динамически получить объекты JavaBean, используя конструктор `ItemSummary`. В JPQL это можно сделать с помощью оператора `new`. В запросах на основе критериев – с помощью метода `construct()`. Класс `ItemSummary` должен иметь конструктор, соответствующий набору элементов в проекции.

Но даже если в классе компонента JavaBean отсутствует нужный конструктор, все равно можно создать его экземпляр и заполнить его с помощью методов записи и/или прямым обращением к полям с применением объекта `AliasToBeanResultTransformer`. Следующий пример преобразует результаты запроса, показанного в листинге 16.1:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
query.setResultTransformer(
    new AliasToBeanResultTransformer(ItemSummary.class)
);
```

```
List<ItemSummary> result = query.list();

for (ItemSummary itemSummary : result) {
    Long itemId = itemSummary.getItemId();
    String name = itemSummary.getName();
    Date auctionEnd = itemSummary.getAuctionEnd();
    // ...
}
```

Конструктору преобразователя нужно передать класс компонента JavaBean; здесь это класс `ItemSummary`. Hibernate требует, чтобы такой класс не имел конструктора или имел общедоступный конструктор без параметров.

Во время преобразования результатов запроса Hibernate отыскивает методы записи и поля, имена которых совпадают с псевдонимами в запросе. В классе `ItemSummary` должны быть определены поля `itemId`, `name` и `auctionEnd` или методы `setItemId()`, `setName()` и `setAuctionEnd()`. Поля и методы должны иметь правильный тип. Если только часть псевдонимов из запроса отображается на поля класса, а оставшаяся часть – на методы записи, это тоже нормально.

Вам также будет полезно узнать, как написать собственный преобразователь `ResultTransformer` на тот случай, если ни один из существующих не подходит.

16.1.4. Создание преобразователя ResultTransformer

Преобразователи, встроенные в Hibernate, довольно простые; между результатами, представленными в виде списков, словарей или массивов объектов, нет особой разницы. Несмотря на то что реализация интерфейса `ResultTransformer` тривиальна, дополнительная логика преобразования результатов запроса может усилить связанность уровней кода приложения. Если код пользовательского интерфейса уже знает, как отображать таблицу на основе списка `List<ItemSummary>`, пусть Hibernate возвращает его напрямую из запроса.

Далее мы покажем, как реализовать преобразователь `ResultTransformer`. Предположим, что требуется получить список `List<ItemSummary>` из запроса в листинге 16.1, но так, чтобы Hibernate не участвовал в создании экземпляров `ItemSummary`, вызывая конструктор через механизм рефлексии. Возможно, класс `ItemSummary` предопределен и не имеет нужного конструктора, полей или методов. Зато у вас есть фабрика `ItemSummaryFactory`, производящая экземпляры `ItemSummary`.

Интерфейс `ResultTransformer` требует реализации методов `transformTuple()` и `transformList()`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/TransformResults.java

```
query.setResultTransformer(
    new ResultTransformer() {

        @Override      ← ❶ Преобразует записи в результате
        public Object transformTuple(Object[] tuple, String[] aliases) {

            Long itemId = (Long) tuple[0];
```

```
            String name = (String) tuple[1];
            Date auctionEnd = (Date) tuple[2];

            assertEquals(aliases[0], "itemId");   ← При необходимости можно получить
            assertEquals(aliases[1], "name");       все псевдонимы запроса
            assertEquals(aliases[2], "auctionEnd");

            return ItemSummaryFactory.newItemSummary(
                itemId, name, auctionEnd
            );
        }

        @Override      ← ❷ Преобразование списка результатов
        public List transformList(List collection) {
            return Collections.unmodifiableList(collection);   ← Коллекция типа
        }                                                          List<ItemSummary>
    }
);
```

❶ Каждый кортеж в результатах запроса, имеющий вид массива `Object[]`, должен быть преобразован в нужный объект. Здесь по индексу из массива извлекается каждый элемент проекции и выполняется вызов фабрики `ItemSummaryFactory` для получения возвращаемого объекта. Hibernate также передает методу список псевдонимов для каждого элемента проекции. Но для данного преобразователя псевдонимы не требуются.

❷ Вы можете обернуть или модифицировать получившийся список результатов после преобразования результатов запроса. Здесь мы сделали возвращаемый список `List` неизменяемым: это отлично подходит для отчета, где никакие данные не должны меняться.

Как показано в примере, преобразование происходит в два этапа: сначала преобразуется каждый возвращаемый кортеж в нужный объект. Затем в вашем распоряжении оказывается целый список `List` этих объектов, который можно обернуть или преобразовать.

Далее мы обсудим другую удобную особенность Hibernate (не имеющей эквивалента в JPA): фильтры коллекций.

Особенности Hibernate

16.2. Фильтрация коллекций

В главе 7 вы узнали, когда следует (а скорее, не следует) отображать коллекции в предметной модели на Java. Самая большая польза от отображения коллекций – более удобный доступ к данным: вы можете вызывать методы `item.getImages()` или `item.getBids()`, чтобы получить доступ ко всем изображениям или ставкам, связанным с данным товаром `Item`. При этом не нужно писать ни запросов JPQL, ни запросов на основе критериев; Hibernate сделает это за вас, как только вы начнете обход элементов коллекции.

Самая очевидная проблема такого подхода: Hibernate будет всегда выполнять один и тот же запрос, извлекая *все* изображения и ставки для товара `Item`. Вы мо-

жете настроить порядок элементов коллекции, но только в статическом отображении. А что делать, если требуется вывести два списка ставок для товара `Item`, упорядоченных по возрастанию и по убыванию? Вы могли бы вернуться к созданию собственных запросов и не вызывать метода `item.getBids()`, но тогда отображение коллекции может не понадобиться.

Вместо этого вы можете использовать нестандартную особенность Hibernate – *фильтры коллекций*, – которая упрощает создание таких запросов с помощью отображаемых коллекций. Предположим, что у вас есть в памяти хранимый экземпляр `Item`, возможно, загруженный с помощью `EntityManager`. Пусть требуется отобразить все ставки (коллекцию `bids`) для данного товара `Item`, а затем наложить на коллекцию `bids` ограничение – выбрать только ставки, сделанные конкретным пользователем `User`. Также требуется отсортировать список в порядке убывания значений `Bid#amount`.

Листинг 16.2 ❖ Фильтрация и упорядочение коллекции

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java

```
Item item = em.find(Item.class, ITEM_ID);
User user = em.find(User.class, USER_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "where this.bidder = :bidder order by this.amount desc"
);

query.setParameter("bidder", user);
List<Bid> bids = query.list();
```

Метод `session.createFilter()` принимает хранимую коллекцию и фрагмент запроса JPQL. Этот фрагмент не должен содержать предложений `select` и `from` – только ограничения в предложениях `where` и `order by`. Псевдоним `this` ссылается на элемент коллекции, в данном случае на экземпляр `Bid`. Созданный фильтр – это обычный запрос `org.hibernate.Query` со связанным параметром, который можно выполнить, вызвав метод `list()`.

Hibernate не выполняет фильтрацию коллекций в памяти приложения. Если во время вызова фильтра коллекция `Item#bids` не была инициализирована, она такой и останется. Более того, фильтры нельзя применять к временным коллекциям и результатам запроса. Их можно применять только к отображаемым коллекциям, на которые ссылаются экземпляры сущностей, управляемые контекстом хранения. Термин *фильтр* в некотором роде вводит в заблуждение, поскольку результатом фильтрации будет другая, совершенно новая коллекция; исходная коллекция никак не изменится.

Ко всеобщему удивлению (включая разработчиков данной функциональности), даже простейшие фильтры могут оказаться полезными. К примеру, пустой запрос можно использовать для постраничного вывода элементов коллекции:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    ""
);

query.setFirstResult(0);  ← Вернет только две ставки
query.setMaxResults(2);
List<Bid> bids = query.list();
```

Здесь Hibernate выполнит запрос, загрузив только два элемента коллекции, начиная с первой строки результата запроса. Обычно вместе с постраничным выводом применяется и упорядочение `order by`.

В фильтрах коллекций не требуется использовать предложения `from`, но вы можете добавить его, если это соответствует вашему стилю. Фильтры коллекций могут даже не возвращать элементов фильтруемой коллекции.

Следующий фильтр возвращает товар `Item`, проданный любым из пользователей, сделавших ставку:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "from Item i where i.seller = this.bidder"
);

List<Item> items = query.list();
```

Используя предложение `select`, можно определить проекцию. Следующий фильтр извлекает имена всех пользователей, сделавших ставки:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "select distinct this.bidder.username order by this.bidder.username asc"
);

List<String> bidders = query.list();
```

Все это очень интересно, но основная причина существования фильтров коллекций – в том, что они позволяют извлекать элементы коллекции без ее инициализации. Для больших коллекций крайне важно добиться хорошей производительности. Следующий запрос извлекает из коллекции `bids` все ставки, сделанные за товар `Item`, которые больше или равны 100:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/FilterCollections.java

```
Item item = em.find(Item.class, ITEM_ID);

org.hibernate.Query query = session.createFilter(
    item.getBids(),
    "where this.amount >= :param"
);

query.setParameter("param", new BigDecimal(100));
List<Bid> bids = query.list();
```

Этот код не инициализирует коллекцию `Item#bids`, но возвращает новую коллекцию.

До появления JPA 2 запросы на основе критериев были доступны лишь в виде нестандартного API в Hibernate. Сегодня стандартные интерфейсы JPA обладают не меньшей мощностью, чем старый `org.hibernate.Criteria`, поэтому он редко бывает нужен. Но есть несколько особенностей, которые доступны только в Hibernate, такие как *запросы по образцу* и возможность встраивания произвольных фрагментов SQL. В следующем разделе приводится краткий обзор интерфейса `org.hibernate.Criteria` и некоторых его уникальных возможностей.

Особенности Hibernate

16.3. Интерфейс запросов на основе критериев в Hibernate

Используя интерфейсы `org.hibernate.Criteria` и `org.hibernate.Example`, можно создавать запросы программно, создавая и объединяя экземпляры `org.hibernate.criterion.*`. Далее вы увидите, как использовать эти интерфейсы и как с их помощью определять выборки, ограничения, соединения и проекции. Мы предполагаем, что вы уже прочли предыдущую главу и знаете, как эти операции транслируются в код SQL. Для всех запросов, показанных здесь, в предыдущей главе можно найти эквивалентный пример, так что при необходимости вы сможете сравнить все три API.

Начнем с самых простых примеров.

16.3.1. Выборка и упорядочение

Следующий запрос загружает все экземпляры `Item`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
org.hibernate.Criteria criteria = session.createCriteria(Item.class);
List<Item> items = criteria.list();
```

Здесь с помощью `Session` создается экземпляр `org.hibernate.Criteria`. Также можно создать отсоединенный запрос `DetachedCriteria`, не связанный с открытым контекстом хранения:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
DetachedCriteria criteria = DetachedCriteria.forClass(Item.class);

List<Item> items = criteria.getExecutableCriteria(session).list();
```

Когда понадобится выполнить этот запрос, «присоедините» его к сеансу `Session`, вызвав метод `getExecutableCriteria()`.

Обратите внимание, что такая возможность поддерживается только в API запросов на основе критериев Hibernate. При работе с JPA понадобится как минимум объект `EntityManagerFactory`, чтобы получить объект `CriteriaBuilder`.

Имеется возможность упорядочить результаты, как в предложении `order by` в JPQL. Следующий запрос загрузит все экземпляры `User`, упорядочив их по имени и фамилии в порядке возрастания:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<User> users =
    session.createCriteria(User.class)
        .addOrder(Order.asc("firstname"))
        .addOrder(Order.asc("lastname"))
        .list();
```

В данном примере используется прием объединения вызовов методов в цепочку; такие методы, как `addOrder()`, возвращают исходный объект `org.hibernate.Criteria`.

Далее мы рассмотрим, как ограничить выбираемые записи.

16.3.2. Ограничения

Следующий запрос возвращает все экземпляры `Item`, свойство `name` которых содержит строку «Foo»:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Item> items =
    session.createCriteria(Item.class)
        .add(Restrictions.eq("name", "Foo"))
        .list();
```

Интерфейс `Restrictions` – это фабрика объектов `Criterion`, которые можно добавлять в объект `Criteria`. Ссылка на атрибуты производится с помощью обычных строк, как, например, `"name"` для `Item#name`.

Также можно организовать поиск подстроки, как в операторе `like` в JPQL. Следующий запрос найдет всех пользователей (экземпляры `User`), имена которых (`username`) начинаются с «j» или «J»:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<User> users =
    session.createCriteria(User.class)
        .add(Restrictions.like("username", "j",
    MatchMode.START).ignoreCase())
        .list();
```

Параметр `MatchMode.START` является эквивалентом шаблона `j%` в JPQL. Также доступны режимы `EXACT`, `END` и `ANYWHERE`.

С помощью точечной нотации можно обращаться к вложенным атрибутам встраиваемых типов, таким как `Address` объекта `User`.

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<User> users =
    session.createCriteria(User.class)
        .add(Restrictions.eq("homeAddress.city", "Some City"))
        .list();
```

Отличительной особенностью `Criteria` API в Hibernate является возможность добавлять в ограничения фрагменты на языке SQL. Следующий запрос найдет всех пользователей (экземпляры `User`), имена которых (`username`) состоят менее, чем из восьми символов:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<User> users =
    session.createCriteria(User.class)
        .add(Restrictions.sqlRestriction(
            "length({alias}.USERNAME) < ?",
            8,
            StandardBasicTypes.INTEGER
        )).list();
```

Hibernate отправит этот фрагмент SQL в базу данных без изменений. Символ подстановки `{alias}` нужен для передачи псевдонима таблицы в итоговом запросе SQL; он всегда ссылается на отображаемую таблицу корневой сущности (`USERS` в данном случае). Здесь также используется позиционный параметр (именованные не поддерживаются), тип которого определяется значением `StandardBasicTypes.INTEGER`.

Расширение системы критериев Hibernate

Система запросов на основе критериев в Hibernate допускает расширение: можно, к примеру, обернуть вызов SQL-функции `LENGTH()` своей реализацией интерфейса `org.hibernate.criterion.Criterion`.

После создания выборки и ограничений нужно добавить в запрос проекцию для описания возвращаемых данных.

16.3.3. Проекция и агрегирование

Следующий запрос возвращает кортежи, включающие идентификатор, имя пользователя (`username`) и домашний адрес (`homeAddress`) для всех пользователей (`User`):

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Object[]> result =
    session.createCriteria(User.class)
        .setProjection(Projections.projectionList()
            .add(Projections.property("id"))
            .add(Projections.property("username"))
            .add(Projections.property("homeAddress"))
        ).list();
```

Результатом этого запроса будет список (`List`) элементов типа `Object[]`, по одному массиву на каждый кортеж. Каждый массив содержит элемент типа `Long` (или типа, заданного для идентификатора пользователя), `String` и `Address`.

Как и при работе с ограничениями, для преобразования элементов проекции можно использовать произвольные выражения SQL и функции:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<String> result =
    session.createCriteria(Item.class)
        .setProjection(Projections.projectionList()
            .add(Projections.sqlProjection(
                "NAME || ':' || AUCTIONEND as RESULT",
                new String[]{"RESULT"},
                new Type[]{StandardBasicTypes.STRING}
            ))
        ).list();
```

Этот запрос вернет список (`List`) строк (`String`), где каждая строка имеет вид «[Имя товара]:[Дата окончания аукциона]». Второй параметр в проекции – псевдонимы, используемые в запросе: они необходимы Hibernate, чтобы прочитать значения из объекта `ResultSet`. Также необходимо указать тип каждого элемента проекции/псевдонима: здесь это `StandardBasicTypes.STRING`.

Hibernate поддерживает группировку и агрегирование. Следующий запрос подсчитает фамилии пользователей:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Object[]> result =
    session.createCriteria(User.class)
```

```
        .setProjection(Projections.projectionList()
            .add(Projections.groupProperty("lastname"))
            .add(Projections.rowCount())
    ).list();
```

Метод `rowCount()` действует подобно функции `count()` в JPQL. Следующий запрос использует агрегирование, чтобы подсчитать среднюю ставку (`Bid`) для каждого товара (`Item`):

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Object[]> result =
    session.createCriteria(Bid.class)
        .setProjection(Projections.projectionList()
            .add(Projections.groupProperty("item"))
            .add(Projections.avg("amount"))
        ).list();
```

Далее вы узнаете, что `Criteria` API позволяет также выполнять соединения.

16.3.4. Соединения

Внутренние соединения связанных сущностей задаются с помощью вложенных объектов `Criteria`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Bid> result =
    session.createCriteria(Bid.class)
        .createCriteria("item")
        .add(Restrictions.isNotNull("buyNowPrice"))
        .createCriteria("seller")
        .add(Restrictions.eq("username", "johndoe"))
        .list();
```

Внутреннее соединение

Запрос вернет все ставки (экземпляры `Bid`) каждого товара (`Item`) с непустым полем `buyNowPrice`, проданного пользователем (`User`) по имени «johndoe». Первое внутреннее соединение по связи `Bid#item` задается вызовом метода `createCriteria("item")` корневого объекта `Criteria`, созданного для класса `Bid`. Вложенный объект `Criteria` определяет путь к связи, для которой выполняется еще одно внутреннее соединение путем вызова `createCriteria("seller")`. На оба соединения наложены ограничения; они объединяются логическим «И» в предложении `where` итогового запроса SQL.

Внутренние соединения также можно выразить с помощью метода `createAlias()` объекта `Criteria`.

Ниже представлен аналогичный запрос:

```
List<Bid> result =
    session.createCriteria(Bid.class)
```

```
        .createCriteria("item")          | Внутреннее
        .createAlias("seller", "s")      | соединение
        .add(Restrictions.and(
            Restrictions.eq("s.username", "johndoe"),
            Restrictions.isNotNull("buyNowPrice")
        ))
        .list();
```

Немедленное динамическое извлечение с помощью *внешнего соединения* задается с помощью метода `setFetchMode()`:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Item> result =
    session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .list();
```

Этот запрос вернет все экземпляры `Item` с коллекциями `bids`, инициализированными в этом же запросе SQL.

Остерегайтесь дубликатов

Так же, как при работе с запросами JPQL и запросами на основе критериев JPA, фреймворк Hibernate может возвращать дубликаты объектов `Item`. См. обсуждение этого феномена в разделе 15.4.5.

Точно так же, как в запросах JPQL и запросах на основе критериев JPA, Hibernate может удалять дубликаты в памяти с помощью операции «distinct»:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Item> result =
    session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
        .list();
```

Здесь видно, как преобразователь `ResultTransformer`, о котором мы говорили ранее в этой главе, может применяться к объекту `Criteria`.

Вы можете извлечь несколько связей/коллекций в одном запросе:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
List<Item> result =
    session.createCriteria(Item.class)
        .createAlias("bids", "b", JoinType.LEFT_OUTER_JOIN)
        .setFetchMode("b", FetchMode.JOIN)
```

```
        .createAlias("b.bidder", "bdr", JoinType.INNER_JOIN)
        .setFetchMode("bdr", FetchMode.JOIN)
        .createAlias("seller", "s", JoinType.LEFT_OUTER_JOIN)
        .setFetchMode("s", FetchMode.JOIN)
        .list();
```

Этот запрос вернет все экземпляры `Item`, загрузит для каждого коллекцию `Item#bids`, используя внешнее соединение, а потом загрузит поле `Bid#bidder`, используя внутреннее соединение. Также будет загружено поле `Item#seller`: поскольку оно не может принимать значения `null`, для его загрузки можно использовать любое соединение. Только не загружайте несколько коллекций в одном запросе, иначе получится декартово произведение (см. раздел 15.4.5).

Далее вы увидите, как с помощью вложенных объектов `Criteria` описываются подзапросы.

16.3.5. Подзапросы

Следующий подзапрос вернет всех пользователей (экземпляры `User`), продающих более одного товара:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
DetachedCriteria sq = DetachedCriteria.forClass(Item.class, "i");
sq.add(Restrictions.eqProperty("i.seller.id", "u.id"));
sq.setProjection(Projections.rowCount());

List<User> result =
    session.createCriteria(User.class, "u")
        .add(Subqueries.lt(1l, sq))
        .list();
```

Экземпляр `DetachedCriteria` описывает запрос, возвращающий количество товаров, проданных данным пользователем (`User`). Поскольку ограничение зависит от псевдонима `u`, это коррелированный подзапрос. «Внешний» запрос, включающий объект `DetachedCriteria`, подставит реальное значение псевдонима `u`. Обратите внимание, что подзапрос является правым операндом оператора `lt()` (меньше, чем), который преобразуется в код SQL: `1 < ([Количество результатов запроса])`.

В Hibernate также можно использовать кванторы. Например, следующий запрос найдет товары со ставками не выше 10:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
DetachedCriteria sq = DetachedCriteria.forClass(Bid.class, "b");
sq.add(Restrictions.eqProperty("b.item.id", "i.id"));
sq.setProjection(Projections.property("amount"));

List<Item> result =
    session.createCriteria(Item.class, "i")
        .add(Subqueries.geAll(new BigDecimal(10), sq))
        .list();
```

И снова позиция операнда говорит о том, что сравнение основано на операторе `geAll()` (больше или равно каждому), который найдет все ставки, не превышающие 10.

Итак, у вас уже есть несколько причин для использования `org.hibernate.Criteria` API. Тем не менее в новых приложениях лучше всего использовать стандартизованные языки запросов JPA. Самой интересной особенностью старого API является возможность встраивания выражений SQL в ограничения и проекции. Другой интересной особенностью Hibernate являются *запросы по образцу*.

16.3.6. Запросы по образцу

Идея запросов по образцу заключается в передаче экземпляра сущности фреймворку Hibernate, который в ответ должен загрузить все экземпляры сущностей, похожие на образец. Это может пригодиться в пользовательском интерфейсе, где имеется сложный экран с настройками, поскольку отпадает необходимость создавать дополнительные классы для хранения поисковых запросов.

Предположим, что в приложении есть форма, позволяющая выполнить поиск пользователей (экземпляры `User`) по фамилии. Вы можете связать поле «фамилия» в форме со свойством `User#lastname` и попросить Hibernate загрузить «похожих» пользователей (экземпляры `User`):

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
User template = new User();                                    ← ❶ Создание пустого экземпляра User
template.setLastname("Doe");

org.hibernate.criterion.Example example = Example.create(template);   ←
example.ignoreCase();                                                  ❷ Создание экземпляра
example.enableLike(MatchMode.START);                                     Example
example.excludeProperty("activated");                          ← ❸ Игнорировать статус активности

List<User> users =                                             ← ❹ Добавить объект Example в качестве ограничения
    session.createCriteria(User.class)
        .add(example)
        .list();
```

❶ Создать «пустой» экземпляр `User`, который будет играть роль образца, установив искомые значения свойств – фамилию «Doe».

❷ На основе образца создать экземпляр `Example`. Этот класс позволит точнее настроить поиск. Здесь используется поиск подстроки и игнорируется регистр символов, поэтому критериям поиска будет соответствовать строка «Doe», «doeX» или «Doe Y».

❸ Класс `User` имеет поле `activated` типа `boolean`. Поскольку простой тип не может принимать значения `null`, его значением по умолчанию будет `false`; поэтому Hibernate включит в круг поиска неактивных пользователей. Но, поскольку нужно найти всех пользователей, следует сообщить Hibernate, чтобы он это поле проигнорировал.

❹ Объект `Example` добавляется как ограничение в запрос, представленный объектом `Criteria`.

Поскольку класс сущности User написан в соответствии с соглашением JavaBean, связать его с интерфейсом пользователя (UI) проще простого. Он имеет необходимые методы чтения/записи, а создать «пустой» экземпляр можно, используя общедоступный конструктор без аргументов (см. обсуждение проектирования конструкторов в разделе 3.2.3).

Очевидным недостатком Example является применение настроек сравнения строк, таких как ignoreCase() и enableLike(), ко *всем* строковым свойствам образца. Если бы мы одновременно искали имя (firstname) и фамилию (lastname), игнорирование регистра символов выполнялось бы для обоих свойств.

По умолчанию в ограничения запроса добавляются все свойства образца сущности, значение которых отлично от null. Как было показано в последнем примере, можно вручную исключить из поиска свойства образца, используя метод excludeProperty(). Аналогично, с помощью метода excludeZeroes(), можно исключить свойства с нулевыми значениями (типа int или long) или отменить любые исключения с помощью метода excludeNone(). В последнем случае любые свойства образца, имеющие значение null, также будут добавлены в ограничение SQL-запроса в виде проверки is null.

Если нужен более полный контроль за включением и исключением свойств, можно расширить класс Example, используя собственный селектор PropertySelector:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
class ExcludeBooleanExample extends Example {
    ExcludeBooleanExample(Object template) {
        super(template, new PropertySelector() {
            @Override
            public boolean include(Object propertyValue,
                                   String propertyName,
                                   Type type) {
                return propertyValue != null
                    && !type.equals(StandardBasicTypes.BOOLEAN);
            }
        });
    }
}
```

Этот селектор исключает любые свойства со значением null (так же, как селектор по умолчанию), но, кроме этого, исключает логические свойства (такие как User#activated).

После добавления образца Example в объект Criteria в виде ограничения можно определить дополнительные ограничения. Также можно создать запрос на основе нескольких образцов. Следующий запрос вернет все товары (экземпляры Item) с названиями (name), начинающимся с «B» или «b», и свойством seller, совпадающим с объектом User:

Файл: /examples/src/test/java/org/jpwh/test/querying/advanced/HibernateCriteria.java

```
Item itemTemplate = new Item();
itemTemplate.setName("B");

Example exampleItem = Example.create(itemTemplate);
exampleItem.ignoreCase();
exampleItem.enableLike(MatchMode.START);
exampleItem.excludeProperty("auctionType");
exampleItem.excludeProperty("createdOn");

User userTemplate = new User();
userTemplate.setLastname("Doe");

Example exampleUser = Example.create(userTemplate);
exampleUser.excludeProperty("activated");

List<Item> items =
    session
        .createCriteria(Item.class)
        .add(exampleItem)
        .createCriteria("seller").add(exampleUser)
        .list();
```

А теперь представьте, сколько кода SQL/JDBC потребовалось бы написать вручную, чтобы создать такой поисковый запрос.

16.4. Резюме

- Вы познакомились с использованием `ResultTransformer` для преобразования результатов запроса и узнали, как вернуть список списков, список словарей и как отобразить псевдонимы в свойства сущностей.
- Мы рассмотрели механизм фильтрации коллекций в Hibernate, помогающий упростить взаимодействия с хранимыми коллекциями.
- Вы познакомились с интерфейсом запросов `Criteria` в Hibernate и узнали, в каких ситуациях он может заменить стандартные запросы на основе критериев JPA. Мы рассмотрели все возможности данного API: выборки и упорядочение, проекции и агрегирование, а также соединения, подзапросы и запросы на основе образцов.

Глава 17

Настройка SQL-запросов

В этой главе:
- назад к JDBC;
- отображение результатов SQL-запросов;
- настройка операций CRUD;
- вызов хранимых процедур.

В этой главе мы рассмотрим настройку SQL-запросов и способы их интеграции с приложениями Hibernate. Язык SQL был создан в 1970 году, но ANSI (American National Standard Institute – Американский институт стандартов) стандартизовал его лишь в 1986-м. Несмотря на то что каждое обновление стандарта SQL добавляло новые возможности (иногда довольно противоречивые), все СУБД поддерживают SQL по-своему. Заботы о переносимости лежат на плечах разработчиков баз данных. И здесь на помощь приходит Hibernate: его встроенный язык запросов генерирует код SQL в соответствии с настроенным диалектом базы данных. Диалекты также влияют на весь автоматически генерируемый код SQL (например, используемый для извлечения коллекции по требованию). Простой заменой диалекта можно интегрировать свое приложение с другой СУБД. Hibernate генерирует все выражения SQL за вас: операции создания, чтения, изменения и удаления (CRUD).

Но иногда может потребоваться нечто большее, чем могут дать Hibernate и Java Persistence API, и тогда приходится спускаться на более низкий уровень абстракции. Используя Hibernate, можно выполнять произвольные выражения SQL:

- можно вернуться к использованию JDBC API и работать напрямую с интерфейсами `Connection`, `PreparedStatement` и `ResultSet`. В Hibernate имеется интерфейс `Connection`, поэтому вам не придется управлять отдельным пулом соединений, и все ваши выражения SQL будут выполняться в рамках одной (текущей) транзакции;
- можно писать SQL-выражения `SELECT`, встраивая их в Java-код или сохраняя в отдельном XML-файле или аннотациях в виде именованных запросов. Эти SQL-запросы будут выполняться с помощью Java Persistence API, как обычные запросы JPQL. Hibernate сможет преобразовать результаты в со-

ответствии с заданным отображением. Этот прием можно также использовать для вызова хранимых процедур;

- SQL-выражения, сгенерированные фреймворком Hibernate, можно заменить своими собственными, написанными вручную. Это значит, что при загрузке экземпляра сущности методом `em.find()` или при загрузке коллекции по требованию данные будет извлекать ваш собственный запрос SQL. Можно даже написать собственный язык управления данными (Data Manipulation Language, DML), включающий такие инструкции, как `UPDATE`, `INSERT` и `DELETE`. А также для выполнения операций CRUD можно вызывать хранимые процедуры. Все автоматически сгенерированные выражения SQL можно заменить собственным кодом.

Мы начнем эту главу с использования обычного JDBC, а затем обсудим возможности Hibernate по отображению результатов запроса. Потом покажем, как в Hibernate переопределяются запросы и DML-операции. И в завершение расскажем об интеграции с хранимыми процедурами в базе данных.

Главные нововведения в JPA 2

- Результаты SQL-запросов можно отображать в вызовы конструкторов.
- С помощью `StoredProcedureQuery` можно вызывать хранимые процедуры и функции непосредственно.

Особенности Hibernate

17.1. Назад к JDBC

Иногда бывает нужно обращаться к базе данных напрямую, с помощью JDBC API и минуя Hibernate. Для этого вам понадобится интерфейс `java.sql.Connection`, с помощью которого можно создать и выполнить запрос (объект `PreparedStatement`), а также напрямую обратиться к результатам запроса (объекту `ResultSet`). Поскольку Hibernate уже знает, как создавать и закрывать соединение с базой данных, он может передать вашему приложению объект `Connection` и автоматически освободить его, когда надобность в нем отпадет.

Эта функциональность доступна благодаря интерфейсу, `org.hibernate.jdbc.Work`, основанному на обратных вызовах. Все взаимодействие с JDBC заключено в реализации этого интерфейса; Hibernate вызовет ее, передав объект `Connection`. Следующий пример выполняет SQL-запрос `SELECT` и производит обход результатов в коллекции `ResultSet`.

Листинг 17.1 ❖ Инкапсуляция взаимодействия с интерфейсами JDBC

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/JDBCFallback.java

```
public class QueryItemWork implements org.hibernate.jdbc.Work {

    final protected Long itemId;  ← ❶ Идентикатор экземпляра Item
```

```
    public QueryItemWork(Long itemId) {
        this.itemId = itemId;
    }

    @Override                          ← ❷ Вызов метода execute()
    public void execute(Connection connection) throws SQLException {
        PreparedStatement statement = null;
        ResultSet result = null;
        try {
            statement = connection.prepareStatement(
                "select * from ITEM where ID = ?"
            );
            statement.setLong(1, itemId);

            result = statement.executeQuery();

            while (result.next()) {
                String itemName = result.getString("NAME");
                BigDecimal itemPrice = result.getBigDecimal("BUYNOWPRICE");
                // ...
            }
        } finally {
            if (result != null)    ← ❸ Освобождение ресурсов
                result.close();
            if (statement != null)
                statement.close();
        }
    }
}
```

❶ Для выполнения этого запроса нужен идентификатор товара, поэтому в классе определено финальное поле, устанавливаемое через параметр конструктора.
❷ Hibernate вызывает метод `execute()` и передает ему объект `Connection`. Вам не нужно закрывать соединение по завершении работы.
❸ Но не забудьте освободить остальные ресурсы, такие как `PreparedStatement` и `ResultSet`.

Выполнить работу, представленную объектом `Work`, можно с помощью `Session`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/JDBCFallback.java

```
UserTransaction tx = TM.getUserTransaction();
tx.begin();
EntityManager em = JPA.createEntityManager();

Session session = em.unwrap(Session.class);
session.doWork(new QueryItemWork(ITEM_ID));

tx.commit();
em.close();
```

В данном случае соединение (`Connection`), возвращаемое Hibernate, будет находиться в границах текущей системной транзакции. Все ваши изменения будут

зафиксированы с подтверждением системной транзакции, а все операции, выполняются они с помощью EntityManager или Session, будут являться частью одной единицы работы. Если вам потребуется вернуть приложению результат выполнения «работы», используйте интерфейс org.hibernate.jdbc.ReturningWork.

Для JDBC-операций, выполняемых внутри реализации интерфейса Work, нет никаких ограничений. Чтобы вызвать хранимую процедуру, вместо PreparedStatement можно использовать CallableStatement; в вашем распоряжении полный доступ к JDBC API.

Для более простых запросов и работы с коллекцией ResultSet, такой как в предыдущем примере, есть более удобная альтернатива.

17.2. Отображение результатов SQL-запросов

После выполнения SQL-запроса SELECT с помощью JDBC API или вызова хранимой процедуры, возвращающей коллекцию ResultSet, выполняется обход записей в результатах и извлекаются нужные данные. Это довольно трудоемкая задача, требующая раз за разом писать один и тот же код.

Быстрая проверка выражений SQL

Для упрощения тестирования сценариев SQL с несколькими СУБД без запуска локального сервера можно воспользоваться онлайн-службой SQL Fiddle по адресу: http://sqlfiddle.com.

Hibernate предлагает альтернативное решение: выполнить обычный SQL-запрос или вызвать хранимую процедуру можно с помощью Hibernate/Java Persistence API, но вместо коллекции ResultSet этот механизм вернет список List с нужными экземплярами. Результат запроса ResultSet можно отобразить в любой класс по вашему усмотрению, а Hibernate сделает все необходимые преобразования.

> **ЗАМЕЧАНИЕ** В этом разделе мы будем говорить только о SQL-запросах SELECT. Однако тот же программный интерфейс можно использовать для выполнения запросов UPDATE и INSERT, как будет показано в разделе 20.1.

Особенности Hibernate

На сегодняшний день существуют два API для выполнения запросов SQL и преобразования их результатов:

- стандартизированный Java Persistence API с методом EntityManager#createNativeQuery() для встроенных SQL-выражений и аннотацией @NamedNativeQuery для запросов, хранящихся в отдельных файлах. Результаты запроса можно отобразить с помощью аннотации @SqlResultSetMapping или определить отображение в JPA-файле orm.xml. Также можно поместить именованные SQL-запросы в XML-файлы JPA;

- нестандартный и более старый механизм Hibernate с методом `Session#createSQLQuery()` и интерфейсом `org.hibernate.SQLQuery` для отображения результатов запроса. Также можно помещать именованные SQL-запросы и отображения результатов в XML-файлы метаданных Hibernate.

Возможности Hibernate шире. Например, он поддерживает немедленную загрузку коллекций и связей сущностей в отображениях результатов SQL-запросов. В следующих разделах мы сравним два API на примере одних и тех же запросов. Начнем с простого встроенного SQL-запроса и отображения скалярного результата проекции.

17.2.1. Проекции в SQL-запросах

Следующий запрос вернет список `List` элементов типа `Object[]`, где каждый элемент представляет кортеж (запись) SQL-проекции:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select NAME, AUCTIONEND from {h-schema}ITEM"
);
List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof String);
    assertTrue(tuple[1] instanceof Date);}
```

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select NAME, AUCTIONEND from {h-schema}ITEM"
);
List<Object[]> result = query.list();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof String);
    assertTrue(tuple[1] instanceof Date);
}
```

Методы `em.createNativeQuery()` и `session.createSQLQuery()` могут принимать обычные запросы SQL в виде строк.

В данном случае запрос извлекает значения столбцов `NAME` и `AUCTIONEND` из таблицы `ITEM`, а Hibernate автоматически преобразует их в значения типа `String` и `java.util.Date`. Для определения типов конкретных элементов фреймворк Hibernate обращается к метаданным `java.sql.ResultSetMetaData`. Ему известно, что тип `VARCHAR` отображается в тип `String`, а `TIMESTAMP` – в тип `java.util.Date` (как объяснялось в разделе 5.3).

Механизм обработки SQL-запросов в Hibernate поддерживает несколько удобных символов подстановки, таких как `{h-schema}` в предыдущем примере. Hibernate заменит символ подстановки схемой, указанной по умолчанию для единицы

хранения (параметр `hibernate.default_schema`). В числе других символов подстановки можно назвать `{h-catalog}` (каталог SQL по умолчанию) и `{h-domain}` (объединяет значения каталога и схемы).

Самое большое преимущество выполнения SQL-выражений с помощью Hibernate состоит в автоматическом преобразовании результатов в экземпляры классов предметной модели.

17.2.2. Отображение в классы сущностей

Следующий запрос SQL вернет список `List` экземпляров сущности `Item`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select * from ITEM",
    Item.class
);

List<Item> result = query.getResultList();
```

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select * from ITEM"
);
query.addEntity(Item.class);

List<Item> result = query.list();
```

Полученные экземпляры `Item` будут находиться в хранимом состоянии под управлением текущего контекста хранения. Он вернет тот же результат, что и JPQL-запрос `select i from Item i`.

Для данного преобразования Hibernate выполнит обход результатов SQL-запроса и попытается отыскать имена и типы столбцов в метаданных отображения. Если столбец `AUCTIONEND` отображается в свойство `Item#auctionEnd`, Hibernate будет знать, как это свойство заполнить, и вернет целиком загруженные экземпляры.

Обратите внимание: Hibernate ожидает, что запрос вернет все столбцы, необходимые для создания экземпляра `Item`, включая свойства, встроенные компоненты и значения столбцов внешних ключей. Если Hibernate не удастся обнаружить отображаемый столбец (по имени) в результате запроса, он возбудит исключение. Чтобы получить такие же имена столбцов, как в метаданных отображения сущности, могут понадобиться псевдонимы в SQL.

Оба интерфейса, `javax.persistence.Query` и `org.hibernate.SQLQuery`, поддерживают связывание параметров. Следующий запрос вернет только один экземпляр сущности `Item`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select * from ITEM where ID = ?",
    Item.class
);
query.setParameter(1, ITEM_ID);  ← Нумерация параметров начинается с 1

List<Item> result = query.getResultList();
```

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select * from ITEM where ID = ?"
);
query.addEntity(Item.class);
query.setParameter(0, ITEM_ID);  ← Нумерация параметров начинается с 0

List<Item> result = query.list();
```

По историческим причинам Hibernate нумерует параметры, начиная с нуля, тогда как JPA – с единицы. По этой причине предпочтительнее использовать именованные параметры:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select * from ITEM where ID = :id",
Item.class
);
query.setParameter("id", ITEM_ID);

List<Item> result = query.getResultList();
```

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select * from ITEM where ID = :id"
);
query.addEntity(Item.class);
query.setParameter("id", ITEM_ID);

List<Item> result = query.list();
```

Несмотря на то что параметризованные запросы поддерживаются в обоих API, спецификация JPA не считает именованные параметры в обычных запросах переносимыми. Следовательно, не все реализации JPA поддерживают именованные параметры в обычных запросах.

Если SQL-запрос возвращает не все столбцы, описанные в отображении класса сущности Java, и его нельзя переписать, используя псевдонимы, следует явно определить отображение результатов запроса.

17.2.3. Настройка отображения запросов

Следующий запрос SQL вернет список `List` управляемых экземпляров сущности `Item`: все столбцы таблицы `ITEM` включены в SQL-проекцию, что требуется для создания экземпляров `Item`. Но столбец `NAME` в проекции переименован в `EXTENDED_NAME` с помощью псевдонима:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select " +
        "i.ID, " +
        "'Auction: ' || i.NAME as EXTENDED_NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID " +
        "from ITEM i",
    "ItemResult"
);

List<Item> result = query.getResultList();
```

Hibernate в этом случае не сможет автоматически сопоставить поля результата запроса со свойствами сущности `Item`: потому что столбца `NAME` больше нет. Следовательно, нужно указать способ отображения результата с помощью второго параметра метода `createNativeQuery()`; в данном случае `ItemResult`.

Отображение полей результата запроса в свойства сущности

Это отображение можно настроить с помощью аннотаций, например в классе `Item`:

Файл: /model/src/main/java/org/jpwh/model/querying/Item.java

```
@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "ItemResult",
        entities =
        @EntityResult(
            entityClass = Item.class,
            fields = {
                @FieldResult(name = "id", column = "ID"),
                @FieldResult(name = "name", column = "EXTENDED_NAME"),
                @FieldResult(name = "createdOn", column = "CREATEDON"),
                @FieldResult(name = "auctionEnd", column = "AUCTIONEND"),
                @FieldResult(name = "auctionType", column = "AUCTIONTYPE"),
                @FieldResult(name = "approved", column = "APPROVED"),
                @FieldResult(name = "buyNowPrice", column = "BUYNOWPRICE"),
```

```
                @FieldResult(name = "seller", column = "SELLER_ID")
            }
        )
    )
})
@Entity
public class Item {
    // ...
}
```

Все поля в результате запроса должны отображаться в свойства класса сущности. Даже если только одно свойство/столбец (`EXTENDED_NAME`) не совпадает с заданным отображением, все равно нужно перечислять все свойства и столбцы.

Отображения результатов запросов SQL, заданные в виде аннотаций, трудно читать, и, кроме того, аннотации JPA можно размещать только перед определением класса, а не в файле метаданных package-info.java. Мы предпочитаем хранить такие отображения в файлах XML. Ниже показано точно такое же отображение:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="ExternalizedItemResult">
    <entity-result entity-class="org.jpwh.model.querying.Item">
        <field-result name="id" column="ID"/>
        <field-result name="name" column="EXTENDED_NAME"/>
        <field-result name="createdOn" column="CREATEDON"/>
        <field-result name="auctionEnd" column="AUCTIONEND"/>
        <field-result name="auctionType" column="AUCTIONTYPE"/>
        <field-result name="approved" column="APPROVED"/>
        <field-result name="buyNowPrice" column="BUYNOWPRICE"/>
        <field-result name="seller" column="SELLER_ID"/>
    </entity-result>
</sql-result-set-mapping>
```

Если отображения будут называться одинаково, приоритет будет отдан отображениям из файла XML.

Сам SQL-запрос можно определить с помощью аннотации `@NamedNativeQuery` или элемента `<namednative-query>`, как было показано в разделе 14.4. В следующих примерах мы будем размещать выражения SQL в Java-коде, поскольку так вам будет проще понять его логику. Но в практике чаще встречаются отображения результатов, задаваемые с помощью более краткого синтаксиса XML.

Давайте сначала повторим предыдущий запрос, используя нестандартный интерфейс Hibernate:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "i.ID as {i.id}, " +
```

```
        "'Auction: ' || i.NAME as {i.name}, " +
        "i.CREATEDON as {i.createdOn}, " +
        "i.AUCTIONEND as {i.auctionEnd}, " +
        "i.AUCTIONTYPE as {i.auctionType}, " +
        "i.APPROVED as {i.approved}, " +
        "i.BUYNOWPRICE as {i.buyNowPrice}, " +
        "i.SELLER_ID as {i.seller} " +
        "from ITEM i"
);
query.addEntity("i", Item.class);

List<Item> result = query.list();
```

В Hibernate отображение результатов запроса можно задать прямо в тексте самого запроса, используя символы подстановки с псевдонимами. Значение псевдонима `i` задается вызовом метода `addEntity()`. Теперь Hibernate сможет сгенерировать настоящие псевдонимы в проекции SQL, используя символы подстановки, такие как `{i.name}` и `{i.auctionEnd}`, указывающие на свойства сущности `Item`. Других определений в отображении результатов запроса не требуется; Hibernate сгенерирует псевдоним в коде SQL и сможет прочитать значения полей коллекции `ResultSet`. Это гораздо удобнее, чем отображать результаты запросов в JPA.

Но если код SQL нельзя изменить, отображения можно определить с помощью методов `addRoot()` и `addProperty()` объекта `org.hibernate.SQLQuery`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "i.ID, " +
        "'Auction: ' || i.NAME as EXTENDED_NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID " +
        "from ITEM i"
);
query.addRoot("i", Item.class)
    .addProperty("id", "ID")
    .addProperty("name", "EXTENDED_NAME")
    .addProperty("createdOn", "CREATEDON")
    .addProperty("auctionEnd", "AUCTIONEND")
    .addProperty("auctionType", "AUCTIONTYPE")
    .addProperty("approved", "APPROVED")
    .addProperty("buyNowPrice", "BUYNOWPRICE")
    .addProperty("seller", "SELLER_ID");

List<Item> result = query.list();
```

Так же, как в стандартном API, в Hibernate можно задать имя существующего отображения:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "i.ID, " +
        "'Auction: ' || i.NAME as EXTENDED_NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID " +
        "from ITEM i"
);
query.setResultSetMapping("ItemResult");

List<Item> result = query.list();
```

Еще одна ситуация, когда приходится задавать собственное отображение результатов запроса, – повторение имен столбцов в результатах запроса SQL.

Отображение повторяющихся полей

Следующий запрос загрузит всех продавцов (поле `seller`) всех товаров (экземпляров `Item`) в одном выражении, соединив таблицы `ITEM` и `USERS`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select " +
        "i.ID as ITEM_ID, " +
        "i.NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID, " +
        "u.ID as USER_ID, " +
        "u.USERNAME, " +
        "u.FIRSTNAME, " +
        "u.LASTNAME, " +
        "u.ACTIVATED, " +
        "u.STREET, " +
        "u.ZIPCODE, " +
        "u.CITY " +
        "from ITEM i join USERS u on u.ID = i.SELLER_ID",
```

```
    "ItemSellerResult"
);
List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof Item);
    assertTrue(tuple[1] instanceof User);
    Item item = (Item) tuple[0];
    assertTrue(Persistence.getPersistenceUtil().isLoaded(item, "seller"));
    assertEquals(item.getSeller(), tuple[1]);
}
```

Фактически это немедленная загрузка связи `Item#seller`. Hibernate знает, что каждая запись содержит поля для экземпляров сущностей `Item` и `User`, связанных внешним ключом `SELLER_ID`.

В данном случае в результатах запроса будут повторяться имена столбцов, соответствующих `i.ID` и `u.ID`. Им присвоены псевдонимы `ITEM_ID` и `USER_ID`, поэтому вы должны настроить отображение результата запроса:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="ItemSellerResult">
    <entity-result entity-class="org.jpwh.model.querying.Item">
        <field-result name="id" column="ITEM_ID"/>
        <field-result name="name" column="NAME"/>
        <field-result name="createdOn" column="CREATEDON"/>
        <field-result name="auctionEnd" column="AUCTIONEND"/>
        <field-result name="auctionType" column="AUCTIONTYPE"/>
        <field-result name="approved" column="APPROVED"/>
        <field-result name="buyNowPrice" column="BUYNOWPRICE"/>
        <field-result name="seller" column="SELLER_ID"/>
    </entity-result>
    <entity-result entity-class="org.jpwh.model.querying.User">
        <field-result name="id" column="USER_ID"/>
        <field-result name="name" column="NAME"/>
        <field-result name="username" column="USERNAME"/>
        <field-result name="firstname" column="FIRSTNAME"/>
        <field-result name="lastname" column="LASTNAME"/>
        <field-result name="activated" column="ACTIVATED"/>
        <field-result name="homeAddress.street" column="STREET"/>
        <field-result name="homeAddress.zipcode" column="ZIPCODE"/>
        <field-result name="homeAddress.city" column="CITY"/>
    </entity-result>
</sql-result-set-mapping>
```

Как и прежде, требуется связать все поля каждой полученной сущности с именами столбцов, даже если от первоначального отображения отличаются только два из них.

Отобразить результаты такого запроса в Hibernate намного проще:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "{i.*}, {u.*} " +
        "from ITEM i join USERS u on u.ID = i.SELLER_ID"
);
query.addEntity("i", Item.class);
query.addEntity("u", User.class);

List<Object[]> result = query.list();
```

Hibernate сгенерирует уникальные псевдонимы для символов подстановки {i.*} и {u.*} в SQL-выражении, поэтому в запросе не будет одинаковых имен столбцов.

В предыдущем отображении результатов запросов JPA вы, должно быть, заметили использование точечной нотации при обращении ко встроенному компоненту `homeAddress` экземпляра `User`. Давайте еще раз рассмотрим этот особый случай.

Отображение полей в свойства компонентов

Класс `User` имеет свойство `homeAddress`, встроенный экземпляр класса `Address`. Следующий запрос загрузит всех пользователей (экземпляры `User`):

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select " +
        "u.ID, " +
        "u.USERNAME, " +
        "u.FIRSTNAME, " +
        "u.LASTNAME, " +
        "u.ACTIVATED, " +
        "u.STREET as USER_STREET, " +
        "u.ZIPCODE as USER_ZIPCODE, " +
        "u.CITY as USER_CITY " +
        "from USERS u",
    "UserResult"
);

List<User> result = query.getResultList();
```

В этом запросе переименованы столбцы `STREET`, `ZIPCODE` и `CITY`, поэтому их нужно вручную отобразить в свойства встроенного компонента:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="UserResult">
    <entity-result entity-class="org.jpwh.model.querying.User">
        <field-result name="id" column="ID"/>
        <field-result name="name" column="NAME"/>
```

```
        <field-result name="username" column="USERNAME"/>
        <field-result name="firstname" column="FIRSTNAME"/>
        <field-result name="lastname" column="LASTNAME"/>
        <field-result name="activated" column="ACTIVATED"/>
        <field-result name="homeAddress.street" column="USER_STREET"/>
        <field-result name="homeAddress.zipcode" column="USER_ZIPCODE"/>
        <field-result name="homeAddress.city" column="USER_CITY"/>
    </entity-result>
</sql-result-set-mapping>
```

Мы уже использовали точечную нотацию несколько раз, когда говорили о встроенных компонентах: обратиться к свойству `street` компонента, на который ссылается свойство `homeAddress`, можно как `homeAddress.street`. Для вложенных встроенных компонентов можно использовать выражения вроде `homeAddress.city.name`, если `City` – не простая строка, а другой встраиваемый класс.

Hibernate также поддерживает точечную нотацию в символах подстановки для свойств компонентов. Ниже показан тот же самый запрос с таким же самым отображением результатов:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "u.ID as {u.id}, " +
        "u.USERNAME as {u.username}, " +
        "u.FIRSTNAME as {u.firstname}, " +
        "u.LASTNAME as {u.lastname}, " +
        "u.ACTIVATED as {u.activated}, " +
        "u.STREET as {u.homeAddress.street}, " +
        "u.ZIPCODE as {u.homeAddress.zipcode}, " +
        "u.CITY as {u.homeAddress.city} " +
        "from USERS u"
);
query.addEntity("u", User.class);

List<User> result = query.list();
```

Немедленное извлечение коллекции в запросе SQL доступно только в Hibernate.

Особенности Hibernate

Немедленное извлечение коллекций

Предположим, что требуется загрузить все экземпляры `Item` в одном SQL-запросе, чтобы при этом для каждого экземпляра `Item` была загружена его коллекция `bids`. Это требует использования в запросе SQL левого внешнего соединения:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(   ← ❶ Соединение таблиц ITEM и BID
    "select " +
        "i.ID as ITEM_ID, " +
        "i.NAME, " +
        "i.CREATEDON, " +
        "i.AUCTIONEND, " +
        "i.AUCTIONTYPE, " +
        "i.APPROVED, " +
        "i.BUYNOWPRICE, " +
        "i.SELLER_ID, " +
        "b.ID as BID_ID," +
        "b.ITEM_ID as BID_ITEM_ID, " +
        "b.AMOUNT, " +
        "b.BIDDER_ID " +
        "from ITEM i left outer join BID b on i.ID = b.ITEM_ID"
);
query.addRoot("i", Item.class)     ← ❷ Отображение столбцов в свойства сущностей
    .addProperty("id", "ITEM_ID")
    .addProperty("name", "NAME")
    .addProperty("createdOn", "CREATEDON")
    .addProperty("auctionEnd", "AUCTIONEND")
    .addProperty("auctionType", "AUCTIONTYPE")
    .addProperty("approved", "APPROVED")
    .addProperty("buyNowPrice", "BUYNOWPRICE")
    .addProperty("seller", "SELLER_ID");

query.addFetch("b", "i", "bids")   ← ❸ Отображение свойств Bid в атрибуты результата запроса
    .addProperty("key", "BID_ITEM_ID")
    .addProperty("element", "BID_ID")
    .addProperty("element.id", "BID_ID")
    .addProperty("element.item", "BID_ITEM_ID")
    .addProperty("element.amount", "AMOUNT")
    .addProperty("element.bidder", "BIDDER_ID");

List<Object[]> result = query.list();

assertEquals(result.size(), 5);    ← ❹ 5 записей в результате

for (Object[] tuple : result) {
    Item item = (Item) tuple[0];   ← ❺ Первый элемент кортежа – экземпляр Item
    assertTrue(Persistence.getPersistenceUtil().isLoaded(item, "bids"));

    Bid bid = (Bid) tuple[1];      ← ❻ Второй элемент – экземпляр Bid
    if (bid != null)
        assertTrue(item.getBids().contains(bid));
}
```

❶ В запросе используется внешнее соединение таблиц `ITEM` и `BID`. В проекции определены все столбцы, необходимые для создания экземпляров `Item` и `Bid`. Повторяющиеся имена столбцов, такие как `ID`, в запросе заменены псевдонимами.
❷ Но по этой причине приходится отображать каждый столбец в соответствующее свойство сущности.
❸ Нужно сконструировать объект `FetchReturn` для коллекции `bids`, указав для сущности-владельца псевдоним `i`, а затем отобразить специальные свойства `key` и `element` в столбец внешнего ключа `BID_ITEM_ID` и идентификатор сущности `Bid`. Затем каждое свойство `Bid` отображается в соответствующее поле в результатах запроса. Некоторые поля отображаются дважды, что требуется Hibernate для инициализации коллекции.
❹ Число записей в результатах запроса является произведением: у одного товара три ставки, у другого – одна ставка, у последнего их нет вовсе, что даст в результате пять кортежей.
❺ Первым элементом каждого кортежа является экземпляр `Item` с инициализированной коллекцией `bids`.
❻ Вторым элементом является экземпляр `Bid`.

Если имена столбцов в запросе SQL совпадают с именами в существующем отображении, можно не отображать результата запроса. В этом случае Hibernate самостоятельно сгенерирует нужные псевдонимы в запросе SQL, используя символы подстановки:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "{i.*}, " +
        "{b.*} " +
        "from ITEM i left outer join BID b on i.ID = b.ITEM_ID"
);
query.addEntity("i", Item.class);
query.addFetch("b", "i", "bids");

List<Object[]> result = query.list();
```

Немедленное извлечение коллекций в запросах SQL доступно только в Hibernate; эта функциональность не стандартизована в JPA.

Ограничения при загрузке коллекций в запросах SQL

Используя `org.hibernate.SQLQuery`, можно извлекать только коллекции, представленные связями *один ко многим* и *многие ко многим*. На момент написания книги Hibernate не поддерживал отображения результатов в коллекции простых или встраиваемых типов. Это означает, что невозможно загрузить коллекцию `Item#images`, используя произвольный запрос SQL и `org.hibernate.SQLQuery`.

До сих пор вы видели, как запросы SQL возвращают управляемые экземпляры сущностей. Но точно так же можно возвращать временные экземпляры любого класса, вызывая нужный конструктор.

Отображение результатов запроса в параметры конструктора

В разделе 15.3.2 мы рассмотрели динамическое создание экземпляров в запросах JPQL и запросах на основе критериев. Похожая функциональность поддерживается для обычных запросов JPA. Следующий запрос вернет список `List` экземпляров `ItemSummary`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select ID, NAME, AUCTIONEND from ITEM",
    "ItemSummaryResult"
);
List<ItemSummary> result = query.getResultList();
```

Отображение `ItemSummaryResult` преобразует каждый столбец результата запроса в соответствующий параметр конструктора `ItemSummary`:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="ItemSummaryResult">
    <constructor-result target-class="org.jpwh.model.querying.ItemSummary">
        <column name="ID" class="java.lang.Long"/>
        <column name="NAME"/>
        <column name="AUCTIONEND"/>
    </constructor-result>
</sql-result-set-mapping>
```

Типы возвращаемых столбцов должны соответствовать типам параметров конструктора; по умолчанию Hibernate выберет для столбца `ID` тип `BigInteger`, поэтому нужно указать для него тип `Long` с помощью атрибута `class`.

Hibernate предоставляет вам выбор. Вы можете указать имя существующего отображения результатов запроса или применить *преобразователь результатов*, как было показано в разделе 16.1 для запросов JPQL:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select ID, NAME, AUCTIONEND from ITEM"
);
// query.setResultSetMapping("ItemSummaryResult");   ← ❶ Использовать существующее отображение

query.addScalar("ID", StandardBasicTypes.LONG);   ← ❷ Отобразить поля как скаляры
query.addScalar("NAME");
query.addScalar("AUCTIONEND");
```

```
query.setResultTransformer(  ← ❸ Применить преобразователь результатов
    new AliasToBeanConstructorResultTransformer(
        ItemSummary.class.getConstructor(
            Long.class,
            String.class,
            Date.class
        )
    )
);

List<ItemSummary> result = query.list();
```

❶ Есть возможность использовать существующее отображение.
❷ С другой стороны, можно отобразить поля, возвращаемые запросом, как скаляры. Без применения преобразователя результатов для каждой записи вы получили бы массив `Object[]`.
❸ Применив встроенный преобразователь, можно превратить массив `Object[]` в экземпляр `ItemSummary`.

Как было показано в разделе 15.3.2, Hibernate может использовать любой конструктор для такого отображения. Например, вместо экземпляров `ItemSummary` можно было бы создать экземпляры `Item`. Они будут находиться во временном или в отсоединенном состоянии, в зависимости от присутствия идентификатора в результатах запроса и в отображении.

Также можно смешивать различные виды отображений результатов запроса или напрямую возвращать скалярные значения.

Скалярные и смешанные отображения

Следующий запрос вернет список `List` массивов `Object[]`, первым элементом в которых будет экземпляр сущности `Item` (товар), а вторым – скаляр, представляющий число ставок за каждый товар:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select " +
        "i.*, " +
        "count(b.ID) as NUM_OF_BIDS " +
        "from ITEM i left join BID b on b.ITEM_ID = i.ID " +
        "group by i.ID, i.NAME, i.CREATEDON, i.AUCTIONEND, " +
        "i.AUCTIONTYPE, i.APPROVED, i.BUYNOWPRICE, i.SELLER_ID",
    "ItemBidResult"
);

List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof Item);
    assertTrue(tuple[1] instanceof Number);
}
```

Поскольку повторяющиеся имена столбцов отсутствуют, отображение выглядит просто:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="ItemBidResult">
    <entity-result entity-class="org.jpwh.model.querying.Item"/>
    <column-result name="NUM_OF_BIDS"/>
</sql-result-set-mapping>
```

В Hibernate можно добавить дополнительное скалярное поле, вызвав метод `addScalar()`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/HibernateSQLQueries.java

```
org.hibernate.SQLQuery query = session.createSQLQuery(
    "select " +
        "i.*, " +
        "count(b.ID) as NUM_OF_BIDS " +
        "from ITEM i left join BID b on b.ITEM_ID = i.ID " +
        "group by i.ID, i.NAME, i.CREATEDON, i.AUCTIONEND, " +
        "i.AUCTIONTYPE, i.APPROVED, i.BUYNOWPRICE, i.SELLER_ID"
);
query.addEntity(Item.class);
query.addScalar("NUM_OF_BIDS");

List<Object[]> result = query.list();

for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof Item);
    assertTrue(tuple[1] instanceof Number);
}
```

Наконец, в одном отображении можно совмещать сущности, конструкторы и скаляры. Следующий запрос вернет управляемый хранимый экземпляр сущности `User`, представляющий продавца (поле `seller`) возвращаемого товара `ItemSummary`. Также вы получите количество ставок за каждый товар:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNativeQuery(
    "select " +
        "u.*, " +
        "i.ID as ITEM_ID, i.NAME as ITEM_NAME, i.AUCTIONEND as
    ITEM_AUCTIONEND, " +
        "count(b.ID) as NUM_OF_BIDS " +
        "from ITEM i " +
        "join USERS u on u.ID = i.SELLER_ID " +
        "left join BID b on b.ITEM_ID = i.ID " +
        "group by u.ID, u.USERNAME, u.FIRSTNAME, u.LASTNAME, " +
        "u.ACTIVATED, u.STREET, u.ZIPCODE, u.CITY, " +
```

```
        "ITEM_ID, ITEM_NAME, ITEM_AUCTIONEND",
    "SellerItemSummaryResult"
);

List<Object[]> result = query.getResultList();
for (Object[] tuple : result) {
    assertTrue(tuple[0] instanceof User);
    assertTrue(tuple[1] instanceof BigInteger);
    assertTrue(tuple[2] instanceof ItemSummary);
}
```

Неправильный порядок результатов: ошибка в Hibernate с номером HHH-8678

Для этого запроса задано следующее отображение:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="SellerItemSummaryResult">

    <entity-result entity-class="org.jpwh.model.querying.User"/>

    <constructor-result target-class="org.jpwh.model.querying.ItemSummary">
        <column name="ID" class="java.lang.Long"/>
        <column name="ITEM_NAME"/>
        <column name="ITEM_AUCTIONEND"/>
    </constructor-result>

    <column-result name="NUM_OF_BIDS"/>
</sql-result-set-mapping>
```

Спецификация JPA гарантирует, что в смешанных отображениях результатов запросов каждый кортеж `Object[]` будет содержать следующие элементы: сначала все отображения `<entityresult>`, затем `<constructor-result>` и, наконец, `<column-result>`. XML-схема JPA гарантирует этот порядок в объявлении отображения; но даже если отобразить элементы в другом порядке с помощью аннотаций (которые не могут обеспечить порядка отображений), результат запроса будет сохранять стандартный порядок. Но помните, что на момент написания книги Hibernate возвращал результаты в неправильном порядке, как было показано в примере выше.

Обратите внимание, что с помощью Hibernate можно использовать это же отображение запроса, указав его имя, как было показано ранее. Если требуется более полный контроль над преобразованием результатов, создайте собственный преобразователь результатов, если, конечно, вы не найдете подходящего встроенного преобразователя.

В завершение вы увидите более сложный пример запроса SQL, объявленного в файле XML.

17.2.4. Размещение обычных запросов в отдельных файлах

Сейчас мы покажем, как объявить запрос SQL в файле XML. В настоящих приложениях с большими запросами SQL не очень удобно читать строки в Java-коде, поэтому вам будет проще хранить запросы SQL в файлах XML. Это также упростит тестирование, поскольку можно копировать и вставлять выражения SQL из файла XML в консоль базы данных SQL.

Вы наверняка обратили внимание, что все примеры запросов SQL в предыдущем разделе были тривиально простыми. Фактически ни один из примеров не требовал применения SQL – в каждом случае можно было бы использовать JPQL. Чтобы сделать следующий пример более интересным, мы напишем запрос, который нельзя выразить на языке JPQL, – только на SQL.

Дерево категорий

Рассмотрим класс `Category` (категория) и связь *многие к одному*, которая ссылается сама на себя, как показано на рис. 17.1.

Это отображение – обычная связь с аннотацией `@ManyToOne` перед столбцом внешнего ключа `PARENT_ID`:

Файл: /model/src/main/java/org/jpwh/model/querying/Category.java

```
@Entity
public class Category {
    @ManyToOne
    @JoinColumn(
        name = "PARENT_ID",
        foreignKey = @ForeignKey(name = "FK_CATEGORY_PARENT_ID")
    )
    protected Category parent;  ← Корневая категория не имеет родителя; столбец может содержать null

    // ...
}
```

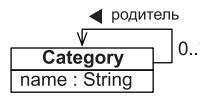


Рис. 17.1 ❖ Сущность `Category` ссылается сама на себя с помощью связи *многие к одному*

Категории образуют дерево. Корнем дерева является категория (экземпляр `Category`) без родителя (свойство `parent`). На рис. 17.2 показан фрагмент такого дерева в базе данных.

CATEGORY

ID	NAME	PARENT_ID
1	One	
2	Two	1
3	Three	1
4	Four	2

Рис. 17.2 ❖ Таблица базы данных, представляющая дерево категорий из примера

Эти данные также можно представить графически, как показано на рис. 17.3. Также эти данные можно представить в виде последовательности путей с указанием уровня каждого узла:

```
/One, 0
/One/Two, 1
/One/Three, 1
/One/Two/Four, 2
```

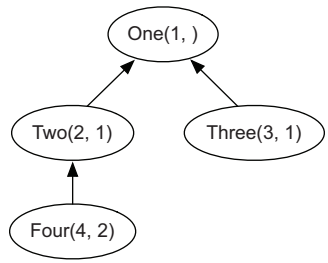


Рис. 17.3 ❖ Дерево категорий

Теперь посмотрим, как приложение загружает экземпляры `Category`. Для этого необходимо найти корневую категорию (экземпляр `Category`). Это можно сделать с помощью простейшего запроса JPQL:

```
select c from Category c where c.parent is null
```

Можно также с легкостью выбрать все категории на конкретном уровне дерева, например всех потомков корневой категории:

```
select c from Category c, Category r where r.parent is null and c.parent = r
```

Этот запрос вернет только прямых потомков корневой категории: `Two` и `Three`.

Но как в одном запросе загрузить дерево (или поддерево) *целиком*? Это невозможно сделать с помощью JPQL, поскольку потребовало бы рекурсии: «Загрузить все категории на данном уровне, затем всех потомков на следующем уровне, потом потомков потомков и т. д.». В SQL такой запрос можно написать, используя *обобщенное табличное выражение* (Common Table Expression, CTE), также известное как *выделение подзапроса*.

Загрузка дерева

Следующий SQL-запрос, объявленный в XML-файле JPA, загрузит все дерево экземпляров `Category`:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<named-native-query name="findAllCategories"
                    result-set-mapping="CategoryResult">
    <query>
```

```
        with CATEGORY_LINK(ID, NAME, PARENT_ID, PATH, LEVEL) as (
            select
                ID,
                NAME,
                PARENT_ID,
                '/' || NAME,
                0
            from CATEGORY where PARENT_ID is null
            union all
            select
                c.ID,
                c.NAME,
                c.PARENT_ID,
                cl.PATH || '/' || c.NAME,
                cl.LEVEL + 1
            from CATEGORY_LINK cl
            join CATEGORY c on cl.ID = c.PARENT_ID
        )
        select
            ID,
            NAME as CAT_NAME,
            PARENT_ID,
            PATH,
            LEVEL
        from CATEGORY_LINK
        order by ID
    </query>
</named-native-query>
```

Это сложный запрос, и мы не будем тратить на него много времени. Чтобы в нем разобраться, прочтите последнее выражение `SELECT`, выбирающее данные из представления `CATEGORY_LINK`. Каждая запись в этом представлении является узлом дерева. Представление определяется с помощью оператора `WITH() AS`. Представление `CATEGORY_LINK` объединяет результаты двух выражений `SELECT`. В процессе рекурсии добавляются дополнительные данные, такие как `PATH` (путь к узлу из корня) и `LEVEL` (уровень узла в дереве).

Давайте отобразим результат этого запроса:

Файл: /model/src/main/resources/querying/NativeQueries.xml

```
<sql-result-set-mapping name="CategoryResult">
    <entity-result entity-class="org.jpwh.model.querying.Category">
        <field-result name="id" column="ID"/>
        <field-result name="name" column="CAT_NAME"/>
        <field-result name="parent" column="PARENT_ID"/>
    </entity-result>
    <column-result name="PATH"/>
    <column-result name="LEVEL" class="java.lang.Integer"/>
</sql-result-set-mapping>
```

Файл XML отображает атрибуты `ID`, `CAT_NAME` и `PARENT_ID` в свойства класса `Category`. Отображение вернет дополнительные скаляры `PATH` и `LEVEL`.

Чтобы выполнить именованный запрос SQL, нужно написать следующий код:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/NativeQueries.java

```
Query query = em.createNamedQuery("findAllCategories");
List<Object[]> result = query.getResultList();

for (Object[] tuple : result) {
    Category category = (Category) tuple[0];
    String path = (String) tuple[1];
    Integer level = (Integer) tuple[2];
    // ...
}
```

Каждый кортеж содержит управляемый хранимый экземпляр `Category`, абсолютный путь к нему от корня дерева (например, `/One`, `/One/Two` и т. д.) и уровень узла.

Этот запрос SQL также можно объявить и отобразить в XML-файле метаданных Hibernate:

Файл: /model/src/main/resources/querying/SQLQueries.hbm.xml

```
<sql-query name="findAllCategoriesHibernate">
    <return class="org.jpwh.model.querying.Category">
        <return-property name="id" column="ID"/>
        <return-property name="name" column="CAT_NAME"/>
        <return-property name="parent" column="PARENT_ID"/>
    </return>
    <return-scalar column="PATH"/>
    <return-scalar column="LEVEL" type="integer"/>
    ...
</sql-query>
```

Мы опустили код SQL, поскольку он такой же, как в примере выше.

Как упоминалось в разделе 14.4, в отношении выполнения кода Java не важно, где определяются именованные запросы – в файлах XML или в аннотациях. Даже язык не играет роли – это может быть как JPQL, так и SQL. Интерфейсы запросов в Hibernate и JPA обладают методами получения именованных запросов и их выполнения, независимо от места, где они определены.

На этом мы заканчиваем рассматривать отображение результатов запросов SQL и переходим к исследованию настройки выражений SQL для операций CRUD, а также способов замены кода SQL, автоматически сгенерированного фреймворком Hibernate, для создания, чтения, изменения и удаления данных из базы.

Особенности Hibernate

17.3. Настройка операций CRUD

Запрос SQL в первом примере ниже загружает экземпляры сущности `User`. Во всех последующих примерах показан код SQL, который Hibernate выполняет автоматически, чтобы вы могли быстрее понять способы отображения.

Настроить способ извлечения экземпляров сущности можно с помощью *загрузчика*.

17.3.1. Подключение собственных загрузчиков

Переопределение запросов SQL для загрузки экземпляров сущности выполняется в два этапа:

- создать именованный запрос, извлекающий экземпляры сущности. Наш пример написан на SQL, но именованные запросы также можно писать на JPQL. В случае SQL может понадобиться дополнительное отображение результатов запроса, как было показано ранее в этой главе;
- активировать запрос, добавив перед классом сущности аннотацию `@org.hibernate.annotations.Loader`. После этого Hibernate будет использовать ваш запрос вместо сгенерированного по умолчанию.

Давайте переопределим способ загрузки экземпляров `User`, как показано в листинге 17.2.

Листинг 17.2 ❖ Загрузка экземпляров `User` с помощью произвольного запроса

Файл: /model/src/main/java/org/jpwh/model/customsql/User.java

```
@NamedNativeQueries({   ← ❶ Объявление запроса для загрузки экземпляров User
    @NamedNativeQuery(
        name = "findUserById",
        query = "select * from USERS where ID = ?",   ← ❷ Символ подстановки
        resultClass = User.class     ← ❸ Дополнительного отображения результатов не требуется
    )
})
@org.hibernate.annotations.Loader(   ← ❹ Активация загрузчика с передачей имени запроса
    namedQuery = "findUserById"
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```

❶ Запрос для загрузки экземпляров `User` задается с помощью аннотаций; его также можно было бы определить в файле XML (с метаданными JPA или Hibernate). Этот запрос можно использовать для доступа к данным, когда потребуется.

❷ У запроса должен иметься только один символ подстановки для значения параметра, который Hibernate заменит значением идентификатора загружаемого экземпляра. Здесь используется позиционный параметр, но именованный тоже подойдет.
❸ Определять отображение результатов для такого простого запроса не нужно. Все поля, возвращаемые запросом, отображаются в классе `User`. Hibernate сможет автоматически преобразовать результат.
❹ После активации загрузчика для класса сущности с определением имени запроса этот запрос будет использоваться для всех операций загрузки экземпляров `User` из базы данных. Здесь нет и намека на язык запроса или способ его объявления; это не зависит от объявления загрузчика.

В именованном запросе загрузчика сущности необходимо выбрать (задать элементы проекции в предложении `SELECT`) следующие свойства класса сущности:

- свойство идентификатора или свойства, входящие в составной первичный ключ;
- скалярные свойства простых типов;
- все свойства, ссылающиеся на встроенные компоненты;
- идентификатор сущности для каждого свойства с аннотацией `@JoinColum` и отображаемой связью, такой как `@ManyToOne`, которой владеет данный класс;
- все скалярные значения, встроенные компоненты и ссылки для соединения по связи, находящиеся внутри аннотации `@SecondaryTable`;
- если для некоторых свойств настроить отложенную загрузку с помощью перехвата вызовов, вам не нужно будет загружать этих свойств (см. раздел 12.1.3).

Hibernate всегда выполняет запрос активного загрузчика, когда требуется извлечь экземпляр `User` из базы данных. Например, ваш запрос будет выполняться при вызове `em.find(User.class, USER_ID)`. Если выполнить цепочку вызовов `someItem.getSeller().getUsername()`, прокси-объект `Item#seller` будет инициализирован с помощью вашего запроса.

Возможно, вам также потребуется настроить операции создания, изменения и удаления экземпляров `User` из базы данных.

17.3.2. Настройка операций создания, изменения, удаления

Как правило, Hibernate генерирует SQL-код для операций CRUD во время запуска. Затем он кэширует выражения SQL для будущего использования, что позволяет во время выполнения не тратить времени на создание выражений SQL для наиболее типичных операций. Пока что вы знаете только, как переопределить букву R (read – чтение) из всей аббревиатуры CRUD, поэтому сейчас мы рассмотрим CUD (создание, изменение, удаление – create, update, delete). Для любой сущности можно определить произвольные SQL-выражения CUD, используя Hibernate-аннотации `@SQLInsert`, `@SQLUpdate` и `@SQLDelete`.

Листинг 17.3 ❖ Замена DML-выражений для сущности `User`

Файл: /model/src/main/java/org/jpwh/model/customsql/User.java

```
@org.hibernate.annotations.SQLInsert(
    sql = "insert into USERS " +
          "(ACTIVATED, USERNAME, ID) values (?, ?, ?)"
)
@org.hibernate.annotations.SQLUpdate(
    sql = "update USERS set " +
          "ACTIVATED = ?, " +
          "USERNAME = ? " +
          "where ID = ?"
)
@org.hibernate.annotations.SQLDelete(
    sql = "delete from USERS where ID = ?"
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```

Будьте внимательны, связывая аргументы SQL-выражений с символами подстановки `?`. Для операций CUD Hibernate поддерживает только позиционные параметры.

Но какой порядок параметров правильный? Существует внутренний порядок связывания аргументов с параметрами SQL-выражений CUD. Чтобы узнать правильный порядок параметров в выражениях SQL, можно позволить Hibernate сгенерировать их примеры. Пока вы еще не добавили собственных выражений SQL, включите уровень журналирования `DEBUG` для категории `org.hibernate.persister.entity`, а затем, после запуска Hibernate, найдите в выводе программы все записи, похожие на следующие:

```
Static SQL for entity: org.jpwh.model.customsql.User
 Insert 0: insert into USERS (activated, username, id) values (?, ?, ?)
 Update 0: update USERS set activated=?, username=? where id=?
 Delete 0: delete from USERS where id=?
```

Эти автоматически сгенерированные выражения SQL подскажут верный порядок параметров, и Hibernate всегда будет выполнять связывание значений в таком же порядке. Скопируйте необходимые выражения SQL и поместите их в аннотации, внеся необходимые изменения.

Особый случай представляют свойства класса сущности, отображаемые в другую таблицу с помощью аннотации `@SecondaryTable`. Настройка выражений CUD до сих пор затрагивала лишь столбцы основной таблицы сущности. Hibernate по-прежнему будет выполнять автоматически сгенерированные выражения SQL для вставки, удаления и изменения строк во вторичной таблице (таблицах). Этот код SQL можно настроить, добавив перед классом сущности аннотацию `@org.hiber-`

nate.annotations.Table и определив ее атрибуты sqlInsert, sqlUpdate и sqlDelete. При желании SQL-выражения CUD можно разместить в файле XML. В этом случае вам останется лишь описать всю сущность целиком в XML-файле метаданных Hibernate. Для произвольных выражений CUD используются элементы <sql-insert>, <sql-update> и <sql-delete>. К счастью, выражения CUD, как правило, гораздо проще запросов выборки, поэтому в большинстве приложений удобнее использовать аннотации.

Теперь мы определили собственные SQL-выражения операций CRUD для экземпляра сущности. Пришла пора переопределить выражения SQL для загрузки и изменения коллекций.

17.3.3. Настройка операций над коллекциями

Давайте переопределим выражения SQL, которые Hibernate использует для загрузки коллекции изображений Item#images. Это коллекция встраиваемых компонентов, отображаемая аннотацией @ElementCollection. К коллекциям базовых типов и связям типа @OneToMany или @ManyToMany применяется та же процедура.

Листинг 17.4 ❖ Загрузка коллекции с помощью собственного запроса

Файл: /model/src/main/java/org/jpwh/model/customsql/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @org.hibernate.annotations.Loader(namedQuery = "loadImagesForItem")
    protected Set<Image> images = new HashSet<Image>();

    // ...
}
```

Как и прежде, нужно объявить запрос для загрузки коллекции. Но на этот раз результаты запроса должны объявляться и отображаться в XML-файле метаданных Hibernate: это единственный способ, позволяющий отобразить результат запроса в свойство поля, представляющее коллекцию:

Файл: /model/src/main/resources/customsql/ItemQueries.hbm.xml

```
<sql-query name="loadImagesForItem">
    <load-collection alias="img" role="Item.images"/>
    select
        ITEM_ID, FILENAME, WIDTH, HEIGHT
    from
        ITEM_IMAGES
    where
        ITEM_ID = ?
</sql-query>
```

Запрос должен иметь единственный (позиционный или именованный) параметр. Hibernate подставит вместо него значение идентификатора сущности, вла-

деющей коллекцией. При каждой инициализации коллекции `Item#images` Hibernate будет выполнять ваш запрос SQL.

Но иногда переопределять весь запрос SQL для загрузки коллекции не требуется, например если нужно всего лишь добавить ограничение в сгенерированное выражение SQL. Предположим, что сущность `Category` содержит коллекцию объектов `Item`, и каждый объект `Item` имеет признак активности. Если свойство `Item#active` имеет значение `false`, при обходе коллекции `Category#items` такой объект загружать не нужно. Такое ограничение можно добавить в запрос SQL с помощью Hibernate-аннотации `@Where`, поместив ее перед отображением коллекции:

Файл: /model/src/main/java/org/jpwh/model/customsql/Category.java

```
@Entity
public class Category {

    @OneToMany(mappedBy = "category")
    @org.hibernate.annotations.Where(clause = "ACTIVE = 'true'")
    protected Set<Item> items = new HashSet<Item>();

    // ...
}
```

Как показано далее, также можно писать собственные запросы SQL для вставки и удаления элементов коллекции.

Листинг 17.5 ❖ Произвольные выражения CUD для модификации коллекции

Файл: /model/src/main/java/org/jpwh/model/customsql/Item.java

```
@Entity
public class Item {

    @ElementCollection
    @org.hibernate.annotations.SQLInsert(
        sql = "insert into ITEM_IMAGES " +
              "(ITEM_ID, FILENAME, HEIGHT, WIDTH) " +
              "values (?, ?, ?, ?)"
    )
    @org.hibernate.annotations.SQLDelete(
        sql = "delete from ITEM_IMAGES " +
              "where ITEM_ID = ? and FILENAME = ? and HEIGHT = ? and WIDTH = ?"
    )
    @org.hibernate.annotations.SQLDeleteAll(
        sql = "delete from ITEM_IMAGES where ITEM_ID = ?"
    )
    protected Set<Image> images = new HashSet<Image>();

    // ...
}
```

Чтобы определить правильный порядок параметров, включите уровень журналирования `DEBUG` для категории `org.hibernate.persister.collection` и найдите

в журнале сгенерированные для этой коллекции выражения SQL; это нужно сделать перед тем, как добавлять аннотации SQL с собственными запросами. Ниже показана новая аннотация `@SQLDeleteAll`, которая может использоваться только для коллекций простых или встраиваемых типов. Hibernate выполнит это выражение SQL, когда потребуется удалить коллекцию из базы данных целиком: например, при вызове `someItem .getImages().clear()` или `someItem.setImages(new HashSet())`.

Для этой коллекции не нужно использовать аннотацию `@SQLUpdate`, поскольку Hibernate не изменяет элементов коллекции встраиваемых типов. Когда меняется значение свойства изображения `Image`, для Hibernate это будет новый объект `Image` в коллекции (не забывайте, что изображения сравниваются «по значению», то есть сравниваются *все* их поля). Поэтому Hibernate выполнит операцию удаления `DELETE` для старого элемента и операцию вставки `INSERT` для нового.

Вместо отложенной загрузки элементов коллекции их можно загружать немедленно, вместе с сущностью-владельцем. Также можно заменить этот запрос собственным выражением SQL.

17.3.4. Немедленное извлечение в собственном загрузчике

Рассмотрим коллекцию ставок `Item#bids` и порядок ее загрузки. По умолчанию Hibernate использует отложенную загрузку, поскольку коллекция отображается аннотацией `@OneToMany`, и, следовательно, запрос для загрузки элементов выполнится, лишь когда начнется обход коллекции. Таким образом, во время загрузки экземпляра `Item` не придется загружать никаких элементов коллекции.

Если, напротив, потребуется загрузить коллекцию `Item#bids` немедленно, вместе с экземпляром `Item`, сначала нужно поместить аннотацию загрузчика с запросом перед определением класса `Item`:

Файл: /model/src/main/java/org/jpwh/model/customsql/Item.java

```
@org.hibernate.annotations.Loader(
    namedQuery = "findItemByIdFetchBids"
)
@Entity
public class Item {

    @OneToMany(mappedBy = "item")
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Как и в предыдущих примерах, необходимо определить именованный запрос в XML-файле метаданных Hibernate, поскольку не существует аннотаций для загрузки коллекций с помощью именованных запросов. Ниже показан код SQL, загружающий экземпляр `Item` вместе с коллекцией `bids` с помощью внешнего соединения (`OUTER JOIN`):

Файл: /model/src/main/resources/customsql/ItemQueries.hbm.xml

```
<sql-query name="findItemByIdFetchBids">
    <return alias="i" class="Item"/>
    <return-join alias="b" property="i.bids"/>
    select
        {i.*}, {b.*}
    from
        ITEM i
    left outer join BID b
        on i.ID = b.ITEM_ID
    where
        i.ID = ?
</sql-query>
```

Вы уже видели этот запрос с отображением результатов в коде Java выше в этой главе, в разделе «Немедленное извлечение коллекций». Здесь добавлено ограничение, допускающее возврат единственной записи из таблицы `ITEM` с заданным первичным ключом.

Аналогично можно немедленно загружать связи с единственным значением, такие как `@ManyToOne`, используя собственные запросы SQL. Предположим, что требуется немедленно загрузить свойство `bidder` вместе с экземпляром `Bid`. Для начала нужно определить загрузчик с именованным запросом:

Файл: /model/src/main/java/org/jpwh/model/customsql/Bid.java

```
@org.hibernate.annotations.Loader(
    namedQuery = "findBidByIdFetchBidder"
)
@Entity
public class Bid {

    @ManyToOne(optional = false, fetch = FetchType.LAZY)
    protected User bidder;

    // ...
}
```

В отличие от запросов, загружающих коллекции, свой именованный запрос можно определить с помощью стандартных аннотаций (конечно, можно поместить его в файл XML, используя синтаксис JPA или Hibernate):

Файл: /model/src/main/java/org/jpwh/model/customsql/Bid.java

```
@NamedNativeQueries({
    @NamedNativeQuery(
        name = "findBidByIdFetchBidder",
        query =
            "select " +
            "b.ID as BID_ID, b.AMOUNT, b.ITEM_ID, b.BIDDER_ID, " +
            "u.ID as USER_ID, u.USERNAME, u.ACTIVATED " +
```

```
            "from BID b join USERS u on b.BIDDER_ID = u.ID " +
            "where b.ID = ?",
        resultSetMapping = "BidBidderResult"
    )
})
@Entity
public class Bid {
    // ...
}
```

Внутреннее соединение (`INNER JOIN`) вполне уместно в данном запросе SQL, поскольку свойство `bidder` экземпляра `Bid` не может быть пустым, а столбец внешнего ключа `BIDDER_ID` не может принимать значения `NULL`. Поскольку в запросе нужно переименовывать повторяющиеся столбцы `ID` в `BID_ID` и `USER_ID`, понадобится собственное отображение результата запроса:

Файл: /model/src/main/java/org/jpwh/model/customsql/Bid.java

```
@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "BidBidderResult",
        entities = {
            @EntityResult(
                entityClass = Bid.class,
                fields = {
                    @FieldResult(name = "id", column = "BID_ID"),
                    @FieldResult(name = "amount", column = "AMOUNT"),
                    @FieldResult(name = "item", column = "ITEM_ID"),
                    @FieldResult(name = "bidder", column = "BIDDER_ID")
                }
            ),
            @EntityResult(
                entityClass = User.class,
                fields = {
                    @FieldResult(name = "id", column = "USER_ID"),
                    @FieldResult(name = "username", column = "USERNAME"),
                    @FieldResult(name = "activated", column = "ACTIVATED")
                }
            )
        }
    )
})
@Entity
public class Bid {
    // ...
}
```

Hibernate выполнит этот запрос SQL и отобразит результаты во время загрузки экземпляра `Bid` при вызове метода `em.find(Bid.class, BID_ID)` или при инициализа-

ции прокси-объекта `Bid`. Hibernate сразу же загрузит поле `Bid#bidder`, невзирая на параметр `FetchType.LAZY` в параметрах связи.

Вот мы и настроили выполнение собственных запросов SQL для всех операций. Далее мы рассмотрим хранимые процедуры и узнаем, как интегрировать их в приложение Hibernate.

17.4. Вызов хранимых процедур

Хранимые процедуры часто используются при разработке приложений баз данных. Размещение кода ближе к данным, и его выполнение внутри базы данных дает определенные преимущества. Это позволяет избежать дублирования функциональности и логики во всех программах, работающих с этими данными. Также принято считать, что общая бизнес-логика тоже не должна дублироваться всеми приложениями. К такой логике относятся процедуры, обеспечивающие целостность данных, проверяющие ограничения, которые слишком сложно описывать декларативно. Вы часто будете встречать в базах данных триггеры, проверяющие правила целостности.

Хранимые процедуры показывают свое преимущество при обработке больших объемов данных для создания отчетов или статистического анализа. Вы должны стараться избегать передачи больших объемов данных между базой и сервером приложения, поэтому при обработке больших объемов следует в первую очередь использовать хранимые процедуры.

Конечно, существуют системы (часто унаследованные), которые даже базовые операции CRUD реализуют с помощью хранимых процедур. Стоит упомянуть и о системах, не позволяющих напрямую выполнять SQL-выражения `INSERT`, `UPDATE` или `DELETE`, а допускающих только вызовы хранимых процедур; когда-то такие системы применялись очень широко (и даже сейчас иногда применяются).

Некоторые СУБД позволяют объявлять пользовательские функции вместо или вместе с хранимыми процедурами. В табл. 17.1 перечислены некоторые различия между процедурами и функциями.

Таблица 17.1. Сравнение процедур и функций, определяемых в базе данных

Хранимая процедура	Функция
Может иметь входные и/или выходные параметры	Может иметь входные параметры
Возвращает ноль, одно или несколько значений	Должна возвращать значение (хотя оно не обязательно будет скаляром или даже `NULL`)
Может вызываться только с помощью JDBC-объекта `CallableStatement`	Может вызываться прямо в предложениях `SELECT`, `WHERE` и т. д.

Трудно обобщать и сравнивать процедуры и функции вне этих очевидных различий. Разные СУБД по-разному их поддерживают: некоторые не поддерживают хранимых процедур или функций, определяемых пользователем, тогда как другие не делают между ними различий (так, например, в PostgreSQL есть только

пользовательские функции). Языки программирования для описания хранимых процедур обычно имеют специфические особенности для каждой СУБД. Некоторые базы данных даже поддерживают хранимые процедуры, написанные на языке Java. Стандартизация хранимых процедур на Java проводилась в рамках стандарта SQLJ, который, к сожалению, не возымел успеха.

В этом разделе мы покажем, как интегрировать хранимые процедуры MySQL и пользовательские функции PostgreSQL с Hibernate. Сначала мы рассмотрим объявления и вызовы хранимых процедур с помощью стандартного Java Persistence API и оригинального Hibernate API. Затем настроим и заменим CRUD-операции Hibernate вызовами хранимых процедур. Важно, чтобы вы прочитали предыдущий раздел перед этим, поскольку интеграция хранимых процедур основана на тех же способах отображения, что и остальные модификации выражений SQL в Hibernate.

Как и ранее в этой главе, хранимые процедуры SQL в примерах довольно просты, чтобы вы могли сосредоточиться на более важных аспектах: вызове процедур и использовании API в своем приложении.

Вызывая хранимую процедуру, вы обычно передаете входные аргументы и получаете возвращаемое значение. Процедуры можно разделить на следующие категории:

- возвращают результат запроса;
- возвращают несколько результатов запросов;
- изменяют данные и возвращают количество измененных строк;
- принимают входные и/или выходные аргументы;
- возвращают курсор, ссылающийся на результат в базе данных.

Рассмотрим простейший случай: хранимую процедуру без параметров, которая возвращает только результат запроса.

17.4.1. Возврат результата запроса

В MySQL можно создать следующую процедуру. Она возвращает результат запроса, содержащий все строки таблицы `ITEM`:

Файл: /model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create procedure FIND_ITEMS()
begin
    select * from ITEM;
end
```

Чтобы вызвать ее, с помощью объекта `EntityManager` нужно создать экземпляр запроса `StoredProcedureQuery` и выполнить его:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "FIND_ITEMS",
    Item.class  ← Или имя отображения результатов запроса
);
```

```
List<Item> result = query.getResultList();
for (Item item : result) {
    // ...
}
```

Как вы уже видели ранее в этой главе, Hibernate автоматически отобразит столбцы возвращаемого результата в свойства класса `Item`. Экземпляры `Item`, возвращаемые этим запросом, будут находиться в управляемом хранимом состоянии. Вместо параметра `Item.class` методу можно передать имя отображения.

Особенности Hibernate

Используя оригинальный интерфейс `Session` фреймворка Hibernate, можно получить результат вызова хранимой процедуры с помощью объекта `Procedure`.

Файл: /example/src/test/java/org/jpwh/test/querying/sql

```
org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("FIND_ITEMS", Item.class);

org.hibernate.result.ResultSetOutput resultSetOutput =
     (org.hibernate.result.ResultSetOutput) call.getOutputs().getCurrent();

List<Item> result = resultSetOutput.getResultList();
```

Метод `getCurrent()` как бы намекает, что процедура может возвращать более одного объекта `ResultSet`. Процедура может вернуть не только несколько результатов, но также количество произведенных изменений, если она изменяла какие-то данные.

17.4.2. Возврат нескольких результатов и количества изменений

Следующая процедура MySQL возвращает все записи из таблицы `ITEM`, которые не были одобрены, меняя их статус `APPROVED`, а также все записи, которые были одобрены:

Файл: /model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create procedure APPROVE_ITEMS()
begin
    select * from ITEM where APPROVED = 0;
    select * from ITEM where APPROVED = 1;
    update ITEM set APPROVED = 1 where APPROVED = 0;
end
```

Приложение в этом случае получит два результата запроса и количество произведенных изменений: доступ к результатам вызова процедуры и их обработка выглядят не так просто, но поскольку JPA тесно связан с JDBC, то, если вы уже работали с хранимыми процедурами, следующий код будет вам знаком:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "APPROVE_ITEMS",
    Item.class                          ← Или имя отображения результатов запроса
);

boolean isCurrentReturnResultSet = query.execute();  ← ❶ Вызов метода execute()
while (true) {                          ← ❷ Обработка результатов вызова
    if (isCurrentReturnResultSet) {     ← ❸ Обработка результатов запроса
        List<Item> result = query.getResultList();
        // ...
    } else {
        int updateCount = query.getUpdateCount();  ← Счетчики изменений закончились:
        if (updateCount > -1) {                      выход из цикла
            // ...
        } else {
            break;                      ← ❹ Обработка счетчиков изменений
        }
    }

    isCurrentReturnResultSet = query.hasMoreResults();  ← ❺ Переход к следующему результату
}
```

❶ Вызов хранимой процедуры методом `execute()`. Он вернет `true`, если хранимая процедура вернула результат запроса, и `false`, если результатом является счетчик изменений.

❷ Обработка всех результатов в цикле. Когда результатов не останется, происходит выход из цикла: в этом случае метод `hasMoreResults()` вернет `false`, а метод `getUpdateCount()` вернет `-1`.

❸ Если процедура вернула результат запроса, обработать его. Hibernate отобразит столбцы каждого результата в управляемые экземпляры класса `Item`. Также можно использовать собственное отображение результатов, применимое для всех результатов, возвращаемых процедурой.

❹ Если текущее возвращаемое значение является счетчиком изменений, метод `getUpdateCount()` вернет значение больше –1.

❺ Метод `hasMoreResults()` выполнит переход к следующему результату и укажет его тип.

Особенности Hibernate

Вызов хранимых процедур с помощью Hibernate может показаться более простым. Фреймворк скрывает сложности, связанные с проверкой типа каждого результата и наличия оставшихся выходных значений процедуры:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("APPROVE_ITEMS", Item.class);

org.hibernate.procedure.ProcedureOutputs callOutputs = call.getOutputs();
```

```
org.hibernate.result.Output output;
while ((output = callOutputs.getCurrent()) != null) {  ← ❶ Проверяет наличие выходных значений
    if (output.isResultSet()) {  ← ❷ Выходное значение – результат запроса?
        List<Item> result =
             ((org.hibernate.result.ResultSetOutput) output)
                .getResultList();
        // ...
    } else {
        int updateCount =  ← ❸ Выходное значение является счетчиком обновлений
             ((org.hibernate.result.UpdateCountOutput) output)
                .getUpdateCount();
        // ...
    }
    if (!callOutputs.goToNext())  ← ❹ Продолжить
        break;
}
```

❶ Пока метод `getCurrent()` не вернул `null`, имеются выходные значения для обработки.
❷ Выходное значение может быть результатом запроса: проверить это и выполнить приведение типа.
❸ Если выходное значение не является результатом запроса, это счетчик обновлений.
❹ Если остались еще выходные значения, продолжить обработку.

Далее мы рассмотрим хранимые процедуры с входными и выходными параметрами.

17.4.3. Передача входных и выходных аргументов

Следующая хранимая процедура MySQL вернет запись из таблицы `ITEM` с заданным идентификатором, а также количество записей в таблице:

Файл: /model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create procedure FIND_ITEM_TOTAL(in PARAM_ITEM_ID bigint,
                                 out PARAM_TOTAL bigint)
begin
    select count(*) into PARAM_TOTAL from ITEM;
    select * from ITEM where ID = PARAM_ITEM_ID;
end
```

Следующая процедура вернет результат запроса с данными из записи `ITEM`. Дополнительно ей передается выходной параметр `PARAM_TOTAL`. Для вызова хранимой процедуры с помощью JPA сначала нужно описать все параметры.

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "FIND_ITEM_TOTAL",
    Item.class
```

```
);
query.registerStoredProcedureParameter(1, Long.class, ParameterMode.IN);
query.registerStoredProcedureParameter(2, Long.class, ParameterMode.OUT);

query.setParameter(1, ITEM_ID);

List<Item> result = query.getResultList();
for (Item item : result) {
    // ...
}
Long totalNumberOfItems = (Long) query.getOutputParameterValue(2);
```

❶ Описание параметров

❷ Связывание значений параметров

❸ Получение результатов

❹ Получение значений выходных параметров

❶ Описание параметров и их типов с их порядковыми номерами (начиная с 1).
❷ Фактические значения связываются с входными параметрами.
❸ Извлекается результат запроса, возвращаемый хранимой процедурой.
❹ После извлечения результата можно прочитать значения выходных параметров.

Также можно описывать и использовать именованные параметры, но позиционные и именованные параметры нельзя смешивать в одном вызове. Обратите еще внимание, что имена параметров в коде Java не обязательно должны совпадать с именами параметров хранимой процедуры. Проще говоря, вы должны описать параметры в том же порядке, как и в определении хранимой процедуры.

Особенности Hibernate

Оригинальный интерфейс Hibernate упрощает описание и использование параметров:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("FIND_ITEM_TOTAL", Item.class);

call.registerParameter(1, Long.class, ParameterMode.IN)
    .bindValue(ITEM_ID);

ParameterRegistration<Long> totalParameter =
    call.registerParameter(2, Long.class, ParameterMode.OUT);

org.hibernate.procedure.ProcedureOutputs callOutputs = call.getOutputs();

org.hibernate.result.Output output;
while ((output = callOutputs.getCurrent()) != null) {
    if (output.isResultSet()) {
        org.hibernate.result.ResultSetOutput resultSetOutput =
             (org.hibernate.result.ResultSetOutput) output;
        List<Item> result = resultSetOutput.getResultList();
        for (Item item : result) {
            // ...
        }
```

❶ Описание параметров

❷ Получение объекта описания параметра

❸ Обработка результатов запроса

```
    }
    if (!callOutputs.goToNext())
        break;
}

Long totalNumberOfItems =
    callOutputs.getOutputParameterValue(totalParameter);
```

❹ Получение значений выходых параметров

❶ Описание всех параметров; можно сразу же выполнить связывание значений.
❷ Описание выходных параметров можно повторно использовать при получении их значений.
❸ Перед получением значений выходных параметров нужно обработать все возвращаемые результаты.
❹ Получение значения выходного параметра с использованием его описания.

Следующая процедура MySQL использует входные параметры для изменения названия товара в записи, в таблице ITEM:

Файл: /model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create procedure UPDATE_ITEM(in PARAM_ITEM_ID bigint,
                             in PARAM_NAME varchar(255))
begin
    update ITEM set NAME = PARAM_NAME where ID = PARAM_ITEM_ID;
end
```

Эта процедура не возвращает результатов запроса – только счетчик изменений, поэтому вызвать ее довольно просто:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "UPDATE_ITEM"
);

query.registerStoredProcedureParameter("itemId", Long.class,
    ParameterMode.IN);
query.registerStoredProcedureParameter("name", String.class,
    ParameterMode.IN);
query.setParameter("itemId", ITEM_ID);
query.setParameter("name", "New Item Name");

assertEquals(query.executeUpdate(), 1);
```

← Счетчик изменений равен 1

```
// Альтернативный вариант:
// assertFalse(query.execute());
// assertEquals(query.getUpdateCount(), 1);
```

← Первое возвращаемое значение– НЕ результат запроса

В этом примере также можно видеть, как работать с именованными параметрами и что имена в коде Java не обязаны совпадать с именами из объявления хранимой процедуры. Но порядок описания параметров по-прежнему важен: PARAM_ITEM_ID должен следовать первым, PARAM_ITEM_NAME – вторым.

Особенности Hibernate

Если вызываемая процедура не возвращает результатов, а лишь изменяет данные, ее вызов можно упростить, использовав метод `executeUpdate()`, который возвращает только счетчик изменений. Также можно последовательно вызвать методы `execute()` и `getUpdateCount()`.

Ниже показан вызов той же процедуры с помощью Hibernate:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("UPDATE_ITEM");

call.registerParameter(1, Long.class, ParameterMode.IN)
    .bindValue(ITEM_ID);

call.registerParameter(2, String.class, ParameterMode.IN)
    .bindValue("New Item Name");

org.hibernate.result.UpdateCountOutput updateCountOutput =
     (org.hibernate.result.UpdateCountOutput) call.getOutputs().getCurrent();

assertEquals(updateCountOutput.getUpdateCount(), 1);
```

Поскольку известно, что процедура не возвращает результатов, можно сразу выполнить приведение первого (текущего) выходного значения к типу `UpdateCountOutput`.

Далее мы рассмотрим случай, когда вместо результата запроса процедура возвращает ссылку на курсор.

17.4.4. Возвращение курсора

В MySQL хранимая процедура не может вернуть курсор. Следующий пример будет работать только с PostgreSQL. Следующая хранимая процедура (или пользовательская функция, поскольку в PostgreSQL – это одно и то же) возвращает курсор для обхода всех записей в таблице `ITEM`:

Файл: /model/src/main/resources/querying/StoredProcedures.hbm.xml

```
create function FIND_ITEMS() returns refcursor as $$
    declare someCursor refcursor;
    begin
        open someCursor for select * from ITEM;
        return someCursor;
    end;
$$ language plpgsql;
```

В JPA курсоры описываются как параметры с помощью специального значения `ParameterMode.REF_CURSOR`:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
StoredProcedureQuery query = em.createStoredProcedureQuery(
    "FIND_ITEMS",
    Item.class
);

query.registerStoredProcedureParameter(
    1,
    void.class,
    ParameterMode.REF_CURSOR
);

List<Item> result = query.getResultList();
for (Item item : result) {
    // ...
}
```

Параметр имеет тип `void`, поскольку его единственная цель состоит в подготовке вызова для дальнейшего чтения данных с помощью курсора. Когда будет вызван метод `getResultList()`, Hibernate сможет вернуть необходимый результат.

Особенности Hibernate

Hibernate также поддерживает автоматическую работу с курсорами:

Файл: /examples/src/test/java/org/jpwh/test/querying/sql/CallStoredProcedures.java

```
org.hibernate.procedure.ProcedureCall call =
    session.createStoredProcedureCall("FIND_ITEMS", Item.class);

call.registerParameter(1, void.class, ParameterMode.REF_CURSOR);

org.hibernate.result.ResultSetOutput resultSetOutput =
     (org.hibernate.result.ResultSetOutput) call.getOutputs().getCurrent();

List<Item> result = resultSetOutput.getResultList();
for (Item item : result) {
    // ...
}
```

Обход результатов запроса с помощью курсора

В разделе 14.3.3 мы обсуждали использование курсоров базы данных для *обхода* потенциально объемных результатов запроса. К сожалению, на момент создания этой книги данная функциональность не поддерживалась ни в JPA, ни в Hibernate. Hibernate будет всегда извлекать в память целиком все результаты запроса, на которые ссылается курсор.

Поддерживать работу с курсорами в различных диалектах СУБД довольно сложно, поэтому Hibernate имеет некоторые ограничения. Например, в PostgreSQL параметр, указывающий на курсор, всегда должен объявляться первым, и (поскольку речь идет о функции) получить из базы данных можно только один курсор. При работе с курсорами в диалекте PostgreSQL Hibernate не поддерживает именованных параметров: вместо них следует использовать позиционные. За более подробной информацией обращайтесь к описанию SQL-диалекта своей базы данных в Hibernate: обратите внимание на такие методы, как `Dialect#getResultSet(CallableStatement)` и т. д.

На этом мы заканчиваем рассмотрение API для вызова хранимых процедур. Далее мы воспользуемся хранимыми процедурами для переопределения сгенерированных выражений, которые Hibernate выполняет для загрузки и сохранения данных.

Особенности Hibernate

17.5. Применение хранимых процедур для операций CRUD

Первой операцией CRUD, которую мы настроим, будет загрузка экземпляров сущностей класса `User`. Ранее в этой главе вы использовали для этого обычный запрос SQL и загрузчик. Если для загрузки экземпляра `User` требуется вызвать хранимую процедуру, сделать это так же просто.

17.5.1. Загрузчик, вызывающий процедуру

Прежде всего нужно создать именованный запрос, вызывающий хранимую процедуру, например с помощью аннотации перед классом `User`:

Файл: /model/src/main/java/org/jpwh/model/customsql/procedures/User.java

```
@NamedNativeQueries({
    @NamedNativeQuery(
        name = "findUserById",
        query = "{call FIND_USER_BY_ID(?)}",
        resultClass = User.class
    )
})
@org.hibernate.annotations.Loader(
    namedQuery = "findUserById"
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```

Сравните его с предыдущей версией в разделе 17.3.1: объявление загрузчика осталось прежним, и ему нужен лишь именованный запрос на любом поддерживаемом языке. Изменился только сам запрос, который также можно поместить в XML-файл метаданных для еще большего разделения функциональности.

JPA не накладывает ограничений на содержимое аннотации `@NamedNativeQuery`: вы можете написать любое выражение SQL. Используя синтаксис экранирования JDBC с фигурными скобками, вы как бы говорите: «Пусть драйвер JDBC сам решает, что с этим делать». Если ваш драйвер JDBC и СУБД умеют работать с хранимыми процедурами, процедуру можно вызвать с помощью конструкции `{call PROCEDURE}`. Hibernate ожидает, что процедура вернет результат запроса, а первая запись в нем будет содержать все столбцы, необходимые для создания экземпляра `User`. Все эти столбцы и свойства были перечислены в разделе 17.3.1. Не забывайте, что всегда можно использовать собственное отображение результатов запроса, если столбцы (их имена), возвращаемые процедурой, не вполне вас устраивают, или когда невозможно изменить код процедуры.

Хранимая процедура должна иметь сигнатуру, позволяющую сделать вызов с единственным аргументом. Hibernate будет использовать этот аргумент как идентификатор при загрузке экземпляра `User`. Ниже показана хранимая процедура в MySQL, обладающая такой сигнатурой:

Файл: /model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure FIND_USER_BY_ID(in PARAM_USER_ID bigint)
begin
    select * from USERS where ID = PARAM_USER_ID;
end
```

Далее мы воспользуемся хранимыми процедурами для создания, изменения и удаления объекта `User`.

17.5.2. Использование процедур в операциях CUD

Для настройки запросов, которые Hibernate использует с целью создания, изменения и удаления экземпляров сущности из базы данных, используются аннотации `@SQLInsert`, `@SQLUpdate`, and `@SQLDelete`. Вместо обычных выражений SQL для этих операций также можно вызывать хранимые процедуры:

Файл: /model/src/main/java/org/jpwh/model/customsql/procedures/User.java

```
@org.hibernate.annotations.SQLInsert(
    sql = "{call INSERT_USER(?, ?, ?)}",
    callable = true
)
@org.hibernate.annotations.SQLUpdate(
    sql = "{call UPDATE_USER(?, ?, ?)}",
    callable = true,
    check = ResultCheckStyle.NONE
```

```
)
@org.hibernate.annotations.SQLDelete(
    sql = "{call DELETE_USER(?)}",
    callable = true
)
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```

Чтобы Hibernate использовал объект JDBC, представляющий вызов процедуры – `CallableStatement` вместо `PreparedStatement`, – нужно указать параметр `callable=true`.

Как объяснялось в разделе 17.3.2, для связывания аргументов в вызове процедуры можно использовать только позиционные параметры, и объявлены они должны быть в том порядке, в каком Hibernate ожидает их увидеть. Хранимые процедуры должны иметь соответствующую сигнатуру. Ниже показано несколько процедур MySQL, которые вставляют, изменяют и удаляют записи из таблицы `USERS`:

Файл: /model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure INSERT_USER(in PARAM_ACTIVATED bit,
                             in PARAM_USERNAME varchar(255),
                             in PARAM_ID bigint)
begin
    insert into USERS (ACTIVATED, USERNAME, ID)
        values (PARAM_ACTIVATED, PARAM_USERNAME, PARAM_ID);
end
```

Файл: /model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure UPDATE_USER(in PARAM_ACTIVATED bit,
                             in PARAM_USERNAME varchar(255),
                             in PARAM_ID bigint)
begin
    update USERS set
        ACTIVATED = PARAM_ACTIVATED,
        USERNAME = PARAM_USERNAME
    where ID = PARAM_ID;
end
```

Файл: /model/src/main/resources/customsql/CRUDProcedures.hbm.xml

```
create procedure DELETE_USER(in PARAM_ID bigint)
begin
    delete from USERS where ID = PARAM_ID;
end
```

После того как хранимая процедура вставит, удалит или изменит экземпляр `User`, Hibernate должен узнать результат ее выполнения. В динамически генери-

руемых SQL-запросах Hibernate проверяет количество измененных строк после операции. Если включено версионирование (см. раздел 11.2.2) и выполняемая операция не смогла обновить ни одной записи, произойдет отказ оптимистической блокировки. Вызывая собственный код SQL, можно это поведение настроить. Хранимая процедура будет сама решать, требуется ли сравнивать текущую версию с версией из базы данных для операций изменения или удаления. Используя параметр аннотации `check`, вы сообщаете Hibernate, как процедура должна реализовать это требование.

По умолчанию используется значение `ResultCheckStyle.NONE`, но также доступны следующие варианты:

- `NONE` – процедура возбудит исключение, если операция завершится неудачей. Hibernate не будет выполнять никаких проверок и полностью положится в этом на процедуру. Если включено версионирование, ваша процедура должна сравнить/увеличить номер версии, а при обнаружении отличий – возбудить исключение;
- `COUNT` – процедура увеличит номер версии, проверит и вернет число измененных записей. Для получения счетчика изменений Hibernate вызовет `CallableStatement#getUpdateCount()`;
- `PARAM` – процедура выполнит увеличение номера версии и проверку, а затем вернет количество измененных записей в первом выходном параметре. В этом случае нужно добавить дополнительный знак вопроса в сигнатуру хранимой процедуры и возвращать число изменившихся записей в этом (первом) выходном параметре. Hibernate автоматически зарегистрирует этот параметр и прочитает его значение после вызова процедуры.

Поддержка параметров ResultCheckStyle

На момент написания книги Hibernate поддерживал только параметр `ResultCheckStyle.NONE`.

И наконец, не забывайте, что Hibernate не всегда может взаимодействовать с хранимыми процедурами и функциями. В таких случаях следует использовать обычный JDBC. Иногда вызов унаследованной хранимой процедуры можно обернуть вызовом новой хранимой процедуры, интерфейс которой будет соответствовать требованиям Hibernate.

17.6. Резюме

- Вы узнали, как можно использовать знакомый JDBC API. Даже при использовании произвольных SQL-запросов Hibernate может взять всю тяжелую работу на себя и преобразовать коллекцию результатов `ResultSet` в экземпляры классов модели предметной области, предоставляя гибкие настройки, включая возможность применения собственного отображения результатов запроса. Для упрощения настроек запросы можно определять во внешних файлах.

- Мы рассмотрели возможности переопределения SQL-выражений для стандартных операций создания, чтения, изменения и удаления (CRUD), а также для операций над коллекциями.
- Вы можете определять собственные загрузчики и использовать в них немедленное извлечение.
- Вы узнали, как вызывать процедуры, хранимые в базе данных, и интегрировать их с Hibernate. Вы узнали, как обрабатывать один или несколько результатов запроса, а также счетчик изменений. Вы узнали, как передаются параметры в хранимые процедуры (входные и выходные) и как они возвращают курсоры базы данных. Также вы узнали, как можно использовать хранимые процедуры для операций CRUD.

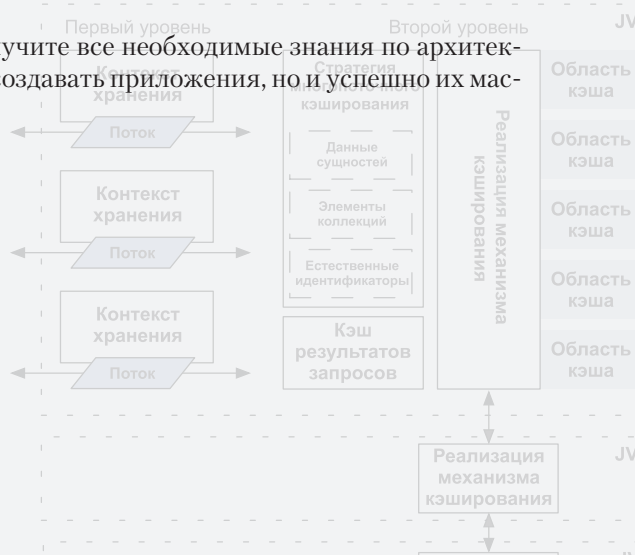
Часть V

СОЗДАНИЕ ПРИЛОЖЕНИЙ

В пятой и последней части в этой книге мы обсудим проектирование и реализацию многоуровневых Java-приложений баз данных, поддерживающих диалоговые взаимодействия. Обсудим наиболее распространенные шаблоны проектирования, используемые с Hibernate, такие как объект доступа к данным (Data Access Object, DAO). Вы увидите, как можно легко протестировать приложение, использующее Hibernate, и какие примы лучше использовать при работе с программным обеспечением для объектно-реляционного отображения (ORM) в веб-приложениях или клиент-серверных приложениях в целом.

Глава 18 целиком посвящена созданию клиент-серверных приложений. Вы познакомитесь с шаблонами клиент-серверной архитектуры, создадите и протестируете уровень хранения, а затем интегрируете экземпляры EJB с JPA. В главе 19 вы изучите создание веб-приложений и способы интеграции JPA с CDI и JSF. Научитесь просматривать данные в таблицах, осуществлять продолжительные диалоговые взаимодействия и настраивать сериализацию сущностей. Наконец, в главе 20 мы покажем возможности масштабирования Hibernate с применением массовых операций и разделяемого кэша.

После прочтения этой части вы получите все необходимые знания по архитектуре, которые позволят вам не только создавать приложения, но и успешно их масштабировать.



Глава 18

Проектирование клиент-серверных приложений

В этой главе:
- шаблоны клиент-серверной архитектуры;
- создание и тестирование уровня хранения;
- интеграция EJB и JPA.

Большинство разработчиков JPA-приложений создает клиент-серверные приложения, основанные на сервере приложений Java с уровнем доступа к базе данных, использующим Hibernate. Зная особенности работы объекта `EntityManager` и системных транзакций, вы наверняка сможете разработать собственную серверную архитектуру. Вы должны будете решить, когда инициализировать объекты `EntityManager`, когда их уничтожать и как определять границы транзакций.

Вас, наверное, интересует, как соотносятся запросы и ответы клиента и сервера с контекстом хранения и транзакциями, протекающими внутри сервера. Должен ли каждый запрос обрабатываться отдельной системной транзакцией? Могут ли несколько последовательных запросов оставлять контекст хранения открытым? Какая роль отводится отсоединенному состоянию сущности? Нужно ли выполнять сериализацию сущности при передаче между клиентом и сервером? Как все эти решения повлияют на проектирование клиента?

Прежде чем ответить на все эти вопросы, мы хотели бы отметить, что в данной главе не будет говориться о других фреймворках приложений, кроме JPA и EJB. Существует несколько причин, из-за которых мы будем использовать JPA вместе EJB во всех примерах:

- наша цель – сосредоточиться на шаблонах проектирования клиент-серверной архитектуры с использованием JPA. Большая часть сквозной функциональности (cross-cutting concerns), такой как сериализация данных при передаче между клиентом и сервером, стандартизована в EJB, поэтому нам

не придется сразу же заниматься решением этой задачи. Мы понимаем, что, скорее всего, вы не станете создавать клиентского приложения с применением EJB. Но благодаря клиентскому коду из этой главы, работающему с EJB, у вас будет основа для принятия решений при выборе и применении другого фреймворка. В следующей главе мы рассмотрим настройку сериализации и покажем, как обмениваться данными в JPA-приложениях с любыми клиентами;

- невозможно охватить все комбинации фреймворков клиентов и серверов из мира Java. Обратите внимание, что мы не ограничились одними веб-приложениями. Конечно, веб-приложения играют важную роль, поэтому мы посвятим следующую главу взаимодействию JPA с JSF и JAX-RS. В этой главе мы рассмотрим клиент-серверное взаимодействие с применением JPA, а также такие абстракции, как *объект доступа к данным* (Data Access Object, DAO), которые будут вам полезны независимо от используемого фреймворка;
- технология EJB очень эффективна, даже при использовании только на стороне сервера. Она позволяет управлять транзакциями и привязывать контекст хранения к сеансовым компонентам с сохранением состояния (stateful session beans). Мы обсудим и эти подробности, так что, если архитектура вашего приложения будет вызывать компоненты EJB на стороне сервера, вы сможете их создать.

В течение этой главы мы реализуем два сценария использования с простой последовательностью действий: редактирование информации о товаре и размещение ставок за товар. Давайте сначала разберемся с уровнем хранения, заключив все операции JPA в один компонент с возможностью его повторного использования: в частности, используя *шаблон объекта доступа к данным*. В результате мы получим прочный фундамент для разработки оставшейся части приложения.

Затем мы реализуем сценарии использования в виде *диалоговых взаимодействий*, представляющих для пользователей приложения законченные единицы работы. Вы увидите код серверных компонентов *с поддержкой состояния* (stateful) и *без поддержки состояния* (stateless), а также поймете, как их применение влияет на проектирование клиентского кода и архитектуру приложения в целом. Выбор компонентов влияет не только на поведение приложения, но и на масштабируемость и устойчивость. Мы реализуем каждый пример с обеими стратегиями, чтобы показать различия между ними.

Итак, приступим к реализации уровня хранения с применением шаблона DAO.

18.1. Разработка уровня хранения

В разделе 3.1.1 мы рассмотрели прием организации отдельного слоя кода, обеспечивающего хранение данных в СУБД. Несмотря на то что JPA поддерживает некоторый уровень абстракции, существует ряд причин, по которым стоит скрывать вызовы JPA за отдельным фасадом:

- собственный уровень хранения может дать более высокий уровень абстракции для операций доступа к данным. Вместо обычных операций CRUD и запросов, выполняемых с помощью объекта `EntityManager`, можно реализовать более высокоуровневые операции, такие как `getMaximumBid(Item i)` (получение самой большой ставки) и `findItems(User soldBy)` (получение всех товаров пользователя). Эта абстракция является главной причиной создания уровня хранения в больших приложениях – она поддерживает повторное использование операций доступа к данным;
- уровень хранения может обладать обобщенным интерфейсом, не раскрывающим подробностей реализации. Другими словами, вы можете скрыть от клиента уровня хранения факт использования Hibernate (или Java Persistence) для доступа к данным. Но мы считаем, что переносимость уровня хранения не является важной характеристикой, поскольку механизмы полного объектно-реляционного отображения, такие как Hibernate, редко обеспечивают переносимость между базами данных. Маловероятно, чтобы вы в будущем захотели переписать ваш уровень хранения, используя другое программное обеспечение, не изменяя при этом клиентского кода. Более того, программный интерфейс Java Persistence является стандартным и полностью переносимым API; ничего страшного, если вы случайно откроете его для клиентов вашего уровня хранения.

Уровень хранения может объединить операции доступа к данным. Эта функциональность связана с переносимостью, но в несколько ином ключе. Представьте, что вам приходится иметь дело с кодом доступа к данным, в котором перемешаны вызовы JPA и JDBC. Создав единый фасад, которым смогут пользоваться клиенты, вы тем самым скроете детали реализации от клиентов. Если вам приходится работать с разными хранилищами данных, это уже хороший повод для создания собственного уровня хранения.

Если вы рассматриваете переносимость и наличие единого интерфейса как побочный эффект создания уровня хранения, вашей главной мотивацией будут предоставление более высокого уровня абстракции и облегчение поддержки и повторного использования кода доступа к данным. Все это достойные причины, поэтому мы рекомендуем создавать уровень хранения с обобщенным фасадом во всех приложениях, кроме самых простых. Но всегда сначала рассматривайте возможность использования JPA без дополнительных уровней. Старайтесь, чтобы код был как можно более простым, добавляя тонкий уровень хранения поверх JPA, только если понимаете, что дублируете некоторые запросы и операции.

Существует множество инструментов, заявляющих, что могут упростить разработку уровня хранения с JPA и Hibernate. Мы советуем сначала работать без подобных инструментов, применяя их, только когда требуется определенная функциональность. Особенно осторожно обращайтесь с генераторами кода и запросов: то, что часто позиционируется как единое решение всех проблем, в долгосрочной перспективе приводит к значительным ограничениям и проблемам сопровождения. Это также может сильно сказаться на продуктивности, если процесс разра-

ботки будет зависеть от работы инструментов генерирования кода. Это касается и собственного инструментария Hibernate – например, если генерировать исходный код класса сущности на основе схемы SQL при каждом внесении изменений. Уровень хранения является важной частью приложения, и, внося дополнительные зависимости, вы должны осознавать последствия. В этой и следующей главах вы увидите, как избежать повторения кода, связанного с компонентами уровня хранения, без использования дополнительных инструментов.

Фасад уровня хранения можно спроектировать несколькими способами – в некоторых маленьких приложениях есть лишь единственный класс для доступа к данным, `DataAccess`; в других операции доступа к данным распределены по классам предметной области, как в шаблоне «активная запись» (Active Record), который не приводится в книге, но мы предпочитаем шаблон DAO.

18.1.1. Обобщенный шаблон «объект доступа к данным»

Шаблон DAO зародился в проекте Java Blueprints компании Sun более 15 лет назад и имеет длинную историю. Класс DAO определяет интерфейс для выполнения операций с конкретной сущностью; согласно этому шаблону весь код, касающийся одной хранимой сущности, должен размещаться в одном месте. Благодаря солидному возрасту накопилось множество вариантов шаблона DAO. Базовая рекомендуемая структура показана на рис. 18.1.

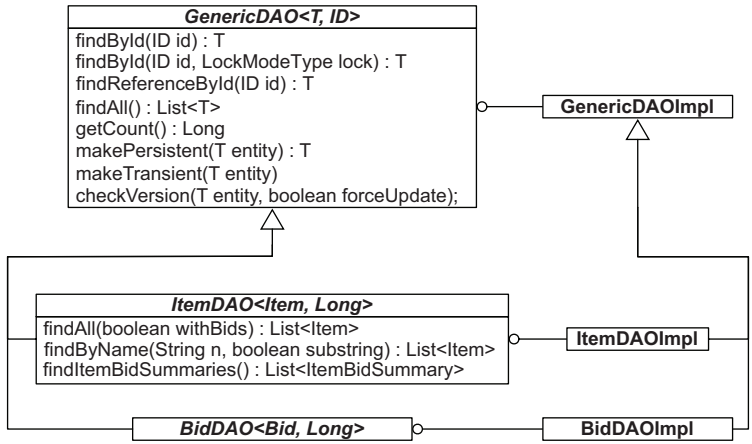


Рис. 18.1 ❖ Обобщенные интерфейсы DAO поддерживают различные реализации

Мы спроектировали уровень хранения с двумя параллельными иерархиями: с одной стороны, интерфейсы, а с другой – их реализации. Базовые операции

сохранения и извлечения объединены в суперинтерфейсе общего назначения и суперклассе, реализующем эти операции, используя конкретное решение (конечно же Hibernate). Интерфейс общего назначения расширяется реализациями для конкретных сущностей, которые требуют дополнительных операций доступа к данным, связанным с бизнес-логикой. Также может иметься несколько реализаций интерфейса DAO для одной сущности.

Давайте взглянем на некоторые интерфейсы и методы, показанные на рис. 18.1. Можно заметить целый набор *методов поиска* с префиксом `find`. Они, как правило, возвращают управляемые сущности (находящиеся в хранимом состоянии), но также могут возвращать произвольные объекты передачи данных, такие как `ItemBidSummary`. Методы поиска создают самую большую проблему дублирования кода; если не проявить осторожности, в программе могут появиться десятки похожих методов. В качестве первого шага следует сделать методы поиска как можно более обобщенными, поместив их в самом верху иерархии (в идеале – в корневом интерфейсе). Возьмем для примера метод `findByName()` в интерфейсе `ItemDAO`: возможно, скоро вам понадобится добавить в него больше параметров поиска, либо сортировать результаты на уровне базы данных, либо реализовать некое подобие постраничной выборки. Мы вернемся к этому позже в разделе 19.2 и покажем, как разработать общее решение для упорядочения и постраничной выборки.

Методы в DAO API ясно указывают, что относятся к уровню хранения, *управляющему состоянием* сущностей. Такие методы, как `makePersistent()` и `makeTransient()`, изменяют состояние экземпляра сущности (или сразу нескольких сущностей, при использовании каскадной передачи состояния). Клиент вправе ожидать, что изменения будут выполняться механизмом хранения автоматически при изменении состояния сущности (поэтому метод `performUpdate()` отсутствует). Если бы интерфейс DAO полагался на SQL-выражения, он бы выглядел совершенно иначе – например, если бы вместо Hibernate использовался обычный JDBC.

Фасад уровня хранения, который мы вам покажем, не открывает клиентам доступа к интерфейсам Hibernate или Java Persistence, поэтому гипотетически можно реализовать его, используя любое программное обеспечение, не оказывая влияния на клиентский код. Возможно, переносимость уровня хранения, о котором мы говорили ранее, для вас не очень важна. В таком случае можно открыть доступ к интерфейсам Hibernate и Java Persistence – например, можно предоставить клиентам доступ к JPA-интерфейсу `CriteriaBuilder` и благодаря этому создать метод общего назначения `findBy(CriteriaQuery)`. Выбор остается за вами; возможно, вы решите, что открыть доступ к интерфейсам Java Persistence безопаснее, чем к интерфейсам Hibernate. Но не стоит забывать, что если еще существует возможность изменения реализации уровня хранения, в зависимости от реализации JPA, то изменить уровень хранения, ориентированный на чистый JDBC, практически невозможно.

Далее мы должны реализовать интерфейсы DAO.

18.1.2. Реализация обобщенных интерфейсов

Рассмотрим возможную реализацию интерфейса `GenericDAO`:

Файл: /apps/app-model/src/main/java/org/jpwh/dao/GenericDAOImpl.java

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    @PersistenceContext
    protected EntityManager em;

    protected final Class<T> entityClass;

    protected GenericDAOImpl(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    // ...
}
```

Для работы обобщенной реализации нужны лишь объект `EntityManager` и класс сущности. Подклассы должны передавать класс сущности через аргумент конструктора. Объект `EntityManager`, напротив, должен предоставляться контейнером среды выполнения, который понимает аннотацию внедрения зависимостей `@PersistenceContext` (например, стандартным контейнером Java EE), или устанавливаться методом `setEntityManager()`.

Далее рассмотрим реализацию методов поиска:

Файл: /apps/app-model/src/main/java/org/jpwh/dao/GenericDAOImpl.java

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    // ...

    public T findById(ID id) {
        return findById(id, LockModeType.NONE);
    }

    public T findById(ID id, LockModeType lockModeType) {
        return em.find(entityClass, id, lockModeType);
    }

    public T findReferenceById(ID id) {
        return em.getReference(entityClass, id);
    }

    public List<T> findAll() {
        CriteriaQuery<T> c =
            em.getCriteriaBuilder().createQuery(entityClass);
```

```
        c.select(c.from(entityClass));
        return em.createQuery(c).getResultList();
    }

    public Long getCount() {
        CriteriaQuery<Long> c =
            em.getCriteriaBuilder().createQuery(Long.class);
        c.select(em.getCriteriaBuilder().count(c.from(entityClass)));
        return em.createQuery(c).getSingleResult();
    }

    // ...
}
```

В коде ясно видно, как класс сущности используется в запросах. Здесь приведены простые запросы на основе критериев, но вы также можете использовать JPQL или SQL.

Наконец, рассмотрим операции для управления состоянием:

Файл: /apps/app-model/src/main/java/org/jpwh/dao/GenericDAOImpl.java

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    // ...

    public T makePersistent(T instance) {
        // merge() handles transient AND detached instances
        return em.merge(instance);
    }

    public void makeTransient(T instance) {
        em.remove(instance);
    }

    public void checkVersion(T entity, boolean forceUpdate) {
        em.lock(
            entity,
            forceUpdate
                ? LockModeType.OPTIMISTIC_FORCE_INCREMENT
                : LockModeType.OPTIMISTIC
        );
    }
}
```

Крайне важно, как реализуется метод `makePersistent()`. Здесь мы решили применить метод `EntityManager#merge()` из-за его универсальности. Если передать ему временный экземпляр сущности, он вернет хранимый экземпляр. При передаче отсоединенного экземпляра сущности сначала будет выполнено слияние, а затем так же возвращен хранимый экземпляр. Это дает клиентам возможность использовать согласованный API, не заботясь о состоянии экземпляра сущности перед вызовом `makePersistent()`. Но при этом клиент должен знать, что объект, возвра-

щаемый методом `makePersistent()`, *всегда* является текущим экземпляром и что аргумент больше не актуален (см. раздел 10.3.4).

На этом завершается создание базового механизма уровня хранения и обобщенного интерфейса, который он предоставляет вышележащим уровням системы. На следующем шаге нужно создать DAO-интерфейсы для каждой сущности, а также их реализацию, путем наследования базового интерфейса и его реализации.

18.1.3. Реализация интерфейсов DAO

Все созданное до сих пор представляет собой абстрактные и обобщенные типы – вы даже не можете создать экземпляра `GenericDAOImpl`. Пришла пора реализовать интерфейс `ItemDAO` путем наследования `GenericDAOImpl` от конкретного класса.

Сначала решим, как клиенты будут вызывать наши интерфейсы DAO. Также необходимо принять во внимание жизненный цикл экземпляра DAO. В данном проекте классы DAO не обладают никаким состоянием, кроме объекта `EntityManager`.

Вызывающие потоки могут совместно использовать один экземпляр DAO. В многопоточном окружении Java EE, к примеру, автоматически внедряемый объект `EntityManager` фактически является потокобезопасным, поскольку внутри он реализован как прокси-объект, использующий контекст хранения, связанный с потоком или транзакцией. Конечно, если вызвать метод `setEntityManager()` интерфейса DAO, такой экземпляр уже нельзя будет совместно использовать в нескольких потоках, а только в одном (например, при интеграционном/модульном тестировании).

Хорошим решением будет использование пула сеансовых компонентов EJB без сохранения состояния; если отметить реализацию `ItemDAOImpl` как компонент EJB без состояния, в него можно внедрить потокобезопасный контекст хранения:

Файл: /apps/app-model/src/main/java/org/jpwh/dao/ItemDAOImpl.java

```
@Stateless
public class ItemDAOImpl extends GenericDAOImpl<Item, Long>
    implements ItemDAO {
    public ItemDAOImpl() {
        super(Item.class);
    }

    // ...
}
```

Совсем скоро вы узнаете, как контейнер EJB выбирает «правильный» контекст хранения для внедрения.

Потокобезопасность внедряемого экземпляра EntityManager

В спецификации Java EE потокобезопасноть экземпляра `EntityManager`, внедряемого с помощью аннотации `@PersistenceContext`, четко не определяется. Спецификация JPA говорит, что обращаться к экземпляру `EntityManager` можно «только из одного потока». Отсюда следует, что один экземпляр не может внедряться в компоненты,

работающие в многопоточной среде, такие как EJB, объекты-одиночки и сервлеты (servlets), поскольку они не работают в однопоточной модели `SingleThreadModel`. Но зато в спецификации EJB говорится, что контейнер EJB должен вызывать каждый сеансовый компонент (с состоянием или без) последовательно. Поэтому каждый экземпляр `EntityManager`, внедряемый в EJB-компонент с состоянием или без, является потокобезопасным; контейнер реализует это поведение за счет внедрения заглушек вместо объектов `EntityManager`. Помимо этого, ваш сервер приложений *может* (но не обязан) поддерживать потокобезопасность доступа к экземпляру `EntityManager`, внедренному в объект-одиночку или многопоточный сервлет. Если у вас остались сомнения, можете внедрить потокобезопасный экземпляр `EntityManagerFactory`, а затем создать собственные экземпляры `EntityManager` в методах служб.

Ниже показаны методы поиска, определенные в интерфейсе `ItemDAO`:

Файл: /apps/app-model/src/main/java/org/jpwh/dao/ItemDAOImpl.java

```
@Stateless
public class ItemDAOImpl extends GenericDAOImpl<Item, Long>
    implements ItemDAO {

    // ...

    @Override
    public List<Item> findAll(boolean withBids) {
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
        // ...
        return em.createQuery(criteria).getResultList();
    }

    @Override
    public List<Item> findByName(String name, boolean substring) {
        // ...
    }

    @Override
    public List<ItemBidSummary> findItemBidSummaries() {
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<ItemBidSummary> criteria =
            cb.createQuery(ItemBidSummary.class);
        // ...
        return em.createQuery(criteria).getResultList();
    }
}
```

После прочтения предыдущих глав у вас не должно было возникнуть проблем с этими запросами, поскольку они довольно простые: воспользуйтесь API запросов на основе критериев или выполняйте именованные JPQL-запросы из внешних файлов. Для работы с запросами на основе критериев следует воспользоваться

статической метамоделью, как было показано в разделе «Использование статической метамодели» в главе 3.

После завершения реализации `ItemDAO` можно браться за `BidDAO`:

Файл: /apps/app-model/src/main/java/org/jpwh/dao/BidDAOImpl.java

```
@Stateless
public class BidDAOImpl extends GenericDAOImpl<Bid, Long>
    implements BidDAO {

    public BidDAOImpl() {
        super(Bid.class);
    }
}
```

Легко видеть, что это «пустая» реализация DAO, которая лишь наследует родительские методы. В следующем разделе мы рассмотрим некоторые операции, которые можно было бы поместить в этот класс DAO. Мы также не приводим здесь кода классов `UserDAO` и `CategoryDAO`, поскольку надеемся, что вы сможете реализовать эти интерфейсы самостоятельно.

Наша следующая тема – тестирование уровня хранения: нужно ли это делать, и если да, то как?

18.1.4. Тестирование уровня хранения

Большая часть примеров в этой книге взята прямо из настоящего тестового кода. Мы продолжим эту традицию в следующих примерах, но должны спросить вас: должны ли вы писать тесты для уровня хранения, чтобы проверить его функциональность?

Исходя из нашего опыта, в тестировании уровня хранения обычно нет смысла. Вы можете создавать экземпляры DAO-классов, внедряя в них фиктивный объект `EntityManager`. Ценность такого модульного теста невелика, а его разработка трудоемка. Вместо этого мы советуем писать *интеграционные* тесты, проверяющие работу большой части приложения с привлечением базы данных. Все оставшиеся примеры в этой главе взяты из таких интеграционных тестов; они имитируют клиента, вызывающего серверное приложение с настоящим кодом для работы с базой данных. Следовательно, сразу проверяется все, что важно: корректная работа служб, бизнес-логика предметной модели, на которую они опираются, и обращение к базе данных через интерфейсы DAO.

Единственная проблема состоит в подготовке такого интеграционного теста. Тестирование должно выполняться в реальном окружении Java EE с настоящим контейнером среды выполнения. Для этих целей мы будем использовать *Arquillian* (http://arquillian.org), инструмент, умеющий взаимодействовать с TestNG. С помощью Arquillian можно создавать в тестовом коде виртуальные архивы и затем запускать на настоящем сервере приложений. Посмотрите на примеры, чтобы понять, как это работает.

Более интересной является задача подготовки данных для интеграционных тестов. Для большей пользы необходимо, чтобы в базе данных была хоть *какая-*

нибудь информация. Нужно загрузить тестовые данные в базу перед началом выполнения тестов, и каждый тест должен работать с правильными, четко определенными данными, чтобы можно было писать надежные утверждения.

Опираясь на наш опыт, выделим три наиболее распространенных способа загрузки тестовых данных:

- инфраструктура тестирования вызывает некоторый метод перед каждым тестом для получения `EntityManager`. Затем вручную инициализируются тестовые сущности и сохраняются в базу данных с помощью `EntityManager`. Главное преимущество данной стратегии в том, что заодно проверяется большое количество отображений. Другим преимуществом является простота программного доступа к тестовым данным. Например, если понадобится значение идентификатора тестового экземпляра `Item`, его можно сразу же получить из метода загрузки данных. Недостатком является сложность поддержки тестовых данных, поскольку их не очень удобно хранить в коде на Java. Очищать тестовые данные в базе можно, удаляя и заново создавая схему после каждого теста, используя механизм Hiberbate для загрузки схемы. До сих пор во всех интеграционных тестах применялся такой подход; вы можете обнаружить процедуры для загрузки данных рядом с каждым тестом в коде примеров;
- Arquillian может импортировать наборы данных с помощью DbUnit (http://dbunit.sourceforge.net) перед запуском каждого теста. DbUnit поддерживает несколько форматов описания наборов данных, в том числе и распространенный «плоский» XML-синтаксис. Это не очень компактно, зато удобно для чтения и сопровождения. Примеры в данной главе используют этот подход. Перед тестовыми классами можно обнаружить Arquillian-аннотацию `@UsingDataSet`, определяющую путь к XML-файлу для загрузки. Hibernate будет генерировать и удалять схему SQL, а Arquillian с помощью DbUnit – выполнять загрузку данных в базу. Если вы предпочитаете хранить тестовые данные отдельно от кода, это решение может вам пригодиться. Если вы не используете Arquillian, не составит труда выполнить загрузку наборов данных вручную, с помощью DbUnit, – посмотрите код `SampleDataImporter` в примерах к этой главе. Мы разворачиваем этот класс на сервере при запуске тестового приложения во время разработки, чтобы у нас были одни и те же данные, как для автоматических тестов, так и для интерактивного взаимодействия;
- в разделе 9.1.1 вы видели, как выполнять собственные сценарии SQL при запуске Hibernate. *Сценарий загрузки* выполняется каждый раз после того, как Hibernate сгенерирует схему; это отличный инструмент для загрузки тестовых данных с помощью обычных `SQL`-выражений INSERT. Этот подход используется в примерах следующей главы. Главным преимуществом является возможность копировать выражения `INSERT` прямо из консоли SQL,

вставляя их в код тестов, и наоборот. Кроме того, если ваша база данных поддерживает синтаксис SQL для описания конструкторов значений, вы сможете создавать компактные инструкции вставки из нескольких строк, как, например, `insert into MY_TABLE (MY_COLUMN) values (1), (2), (3), ...`.

Выбор стратегии остается за вами. Часто выбор зависит от личных предпочтений и объема тестовых данных. Обратите внимание, что мы говорим о тестовых данных для интеграционного тестирования, но не для нагрузочного или проверки масштабирования. Если понадобятся большие объемы тестовых данных (в основном случайных), обратите внимание на такие генераторы данных, как *Benerator* (http://databene.org/databene-benerator.html).

На этом первый этап разработки уровня хранения завершен. Теперь вы можете получать экземпляры `ItemDAO` и обращаться к базе данных на более высоком уровне абстракции. Давайте напишем клиентский код, который будет обращаться к уровню хранения, и реализуем оставшуюся часть приложения.

18.2. Создание сервера без состояния

В качестве приложения мы создадим сервер без состояния, т. е. сервер, не сохраняющий никаких данных между несколькими запросами клиента. Приложение будет простым – мы реализуем только два варианта использования: редактирование данных о товаре и размещение ставки.

Работа с системой будет протекать в *диалоговом режиме*: с точки зрения пользователя это будет одна единица работы. Разработчики не обязательно рассматривают систему так же, как пользователи; как правило, единицей работы для разработчиков является системная транзакция. Сейчас мы подробно рассмотрим это различие и то, как ожидания пользователя влияют на проектирование серверного и клиентского приложений. Начнем с реализации первого диалогового взаимодействия – редактирования информации о товаре.

18.2.1. Редактирование информации о товаре

Роль клиента будет играть простейшее консольное EJB-приложение. На рис. 18.2 показано диалоговое взаимодействие «редактирование информации о товаре», как оно выглядит при работе с этим клиентским приложением.

Клиентское приложение выводит список товаров; пользователь выбирает один из них. Затем клиентское приложение запрашивает у пользователя название операции, которую тот хотел бы выполнить. Наконец, после ввода названия операции клиентское приложение выводит запрос на подтверждение. Теперь система готова для следующего диалогового взамодействия. Клиентское приложение снова выводит список товаров.

Последовательность вызовов в данном диалоговом взаимодействии показана на рис. 18.3. Этот рисунок станет вашим путеводителем по текущему разделу.

```
>>> Starting dialog, connecting to server (press CTRL+C to exit)...
------------------------------------------------------------------------
ID   | Name                | Auction End          |     Highest Bid
------------------------------------------------------------------------
1    | Baseball Glove      | 06. Mar 2015 15:00   |            13.00
2    | Aquarium            | 07. Mar 2015 16:00   |               -
3    | Golf GTI            | 08. Mar 2015 09:30   |         30000.00
4    | Blade Runner Bluray | 09. Mar 2015 10:20   |               -
5    | Coffee Machine      | 10. Mar 2015 14:55   |             6.00
------------------------------------------------------------------------
Please enter an item ID:
1
Would you like to rename (n) the item or place a bid (b):
n
New name for item 'Baseball Glove':
Pretty Baseball Glove
=> Item name changed successfully!
```

Рис. 18.2 ❖ С точки зрения пользователя данное диалоговое взаимодействие является единицей работы

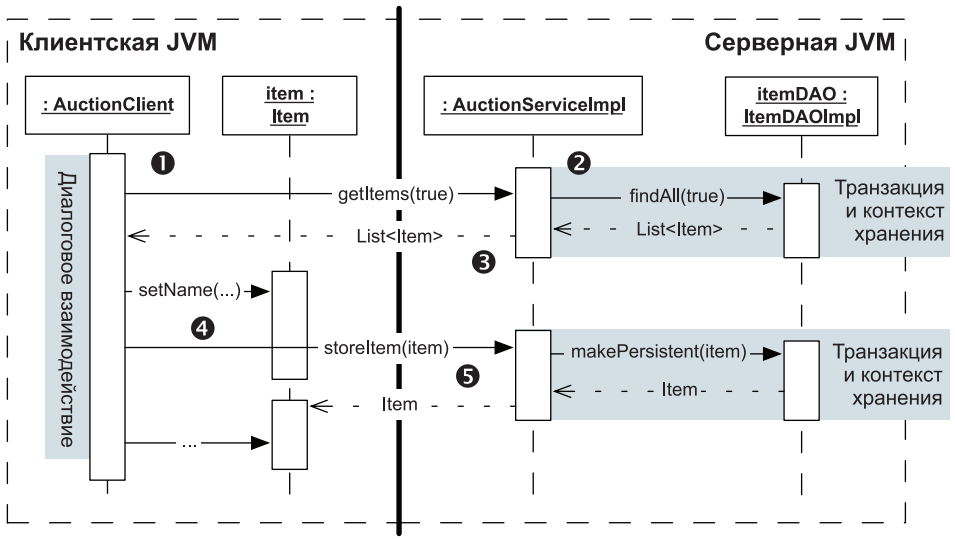


Рис. 18.3 ❖ Вызовы во время диалогового взаимодействия «редактирование информации о товаре»

Давайте внимательнее рассмотрим код; чтобы не сбиваться с нити повествования, обращайте внимание на номера на рисунке. Код, показанный ниже, взят из тестового примера, который имитирует клиентское приложение; далее будет показан серверный код, обрабатывающий клиентские запросы.

Клиентское приложение получает список экземпляров `Item` от сервера, чтобы начать диалоговое взаимодействие ❶; значение `true` в параметре обеспечивает немедленную загрузку коллекции `Item#bids` каждого экземпляра. Поскольку сервер

не хранит промежуточного состояния диалога, эту работу должно взять на себя клиентское приложение:

Файл: /apps/app-stateless-server/src/test/java/org/jpwh/test/stateless/AuctionServiceTest.java

```
List<Item> items;  ← Состоянием приложения – списком товаров – должен управлять клиент

items = service.getItems(true);  ← Получить все товары в отсоединенном состоянии
                                   вместе со ставками
```

Серверный код обработает запрос при помощи DAO:

Файл: /apps/app-stateless-server/src/main/java/org/jpwh/stateless/AuctionServiceImpl.java

```
@javax.ejb.Stateless
@javax.ejb.Local(AuctionService.class)
@javax.ejb.Remote(RemoteAuctionService.class)
public class AuctionServiceImpl implements AuctionService {

    @Inject
    protected ItemDAO itemDAO;

    @Inject
    protected BidDAO bidDAO;

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)  ← Значение по умолчанию
    public List<Item> getItems(boolean withBids) {
        return itemDAO.findAll(withBids);
    }

    // ...

}
```

(Не обращайте внимания на объявленные здесь интерфейсы; они необходимы для удаленных вызовов и локального тестирования компонентов EJB.) Поскольку при вызове `getItems()` ни одна транзакция не активна, будет запущена новая транзакция. При выходе из метода она подтверждается автоматически. Аннотация `@TransactionAttribute` в данном случае не обязательна; по умолчанию методы компонентов EJB должны вызываться в транзакции.

Чтобы получить коллекцию `List` экземпляров `Item`, метод `getItems()` компонента EJB обращается к `ItemDAO` ❷. Контейнер Java EE автоматически выполнит поиск и внедрение экземпляра `ItemDAO`, при этом объект `EntityManager` уже будет внедрен в DAO. Поскольку с текущей транзакцией еще не связан никакой экземпляр `EntityManager` или контекст хранения, будет открыт и присоединен к транзакции новый контекст хранения. Выталкивание контекста и его закрытие произойдут при подтверждении транзакции. Это удобная особенность компонентов EJB без состояния: для работы с JPA в транзакции не требуется особых усилий.

Коллекция `List` отсоединенных экземпляров `Item` (после закрытия контекста хранения) возвращается клиенту ❸. Сейчас вы не должны беспокоиться о сериализации; пока объекты `List` и `Item`, а также все достижимые типы наследуют интерфейс `Serializable`, контейнер EJB сам позаботится об этом.

Затем клиент меняет имя выбранного экземпляра `Item` и просит сервер сохранить изменение, послав измененный и отсоединенный объект `Item` ❹:

Файл: /apps/app-stateless-server/src/test/java/org/jpwh/test/stateless/AuctionServiceTest.java

```
detachedItem.setName("Pretty Baseball Glove");

detachedItem = service.storeItem(detachedItem);
```

Вызов службы и сохранение изменений. Вернется текущий экземпляр Item

Сервер примет отсоединенный экземпляр `Item` и попросит `ItemDAO` сохранить изменения ❺, выполнив слияние:

Файл: /apps/app-stateless-server/src/main/java/org/jpwh/stateless/AuctionServiceImpl.java

```
public class AuctionServiceImpl implements AuctionService {

    // ...

    @Override
    public Item storeItem(Item item) {
        return itemDAO.makePersistent(item);
    }

    // ...
}
```

Обновленное в результате слияния состояние вернется клиенту.

Диалоговое взаимодействие завершено, и клиент может проигнорировать обновленный экземпляр `Item`. Но клиентскому приложению также известно, что возвращаемое значение находится в актуальном состоянии, а все сохраненные до этого состояния, такие как коллекция `List` экземпляров `Item`, являются устаревшими и от них желательно избавиться. Следующее диалоговое взаимодействие должно начаться с новым состоянием, учитывающим последний полученный экземпляр `Item`, а для этого следует получить свежий список.

Теперь вы знаете, как реализовать одно диалоговое взаимодействие – единицу работы с точки зрения пользователя, – используя две системные транзакции на стороне сервера. Поскольку вы загрузили данные в первой системной транзакции, отложив запись изменений до второй транзакции, диалоговое взаимодействие получилось атомарным: изменения не сохранятся, пока последний этап не завершится успехом. Рассмотрим это подробнее на примере второго варианта использования – размещения ставки.

18.2.2. Размещение ставки

Диалоговое взаимодействие по размещению ставки в консольном приложении показано на рис. 18.4. Клиентское приложение выводит список товаров и просит

пользователя выбрать один из них. Пользователь может сделать ставку и получить подтверждение, если ставка была успешно сохранена. Последовательность вызовов и ключевые пункты в коде показаны на рис. 18.5.

```
>>> Starting dialog, connecting to server (press CTRL+C to exit)...
------------------------------------------------------------------------
ID   | Name                | Auction End          |     Highest Bid
------------------------------------------------------------------------
1    | Baseball Glove      | 06. Mar 2015 15:00   |           13.00
2    | Aquarium            | 07. Mar 2015 16:00   |               -
3    | Golf GTI            | 08. Mar 2015 09:30   |        30000.00
4    | Blade Runner Bluray | 09. Mar 2015 10:20   |               -
5    | Coffee Machine      | 10. Mar 2015 14:55   |            6.00
------------------------------------------------------------------------
Please enter an item ID:
1
Would you like to rename (n) the item or place a bid (b):
b
Your bid for item 'Baseball Glove':
15
=> Bid placed successfully!
```

Рис. 18.4 ❖ Пользователь делает ставку: единица работы с его точки зрения

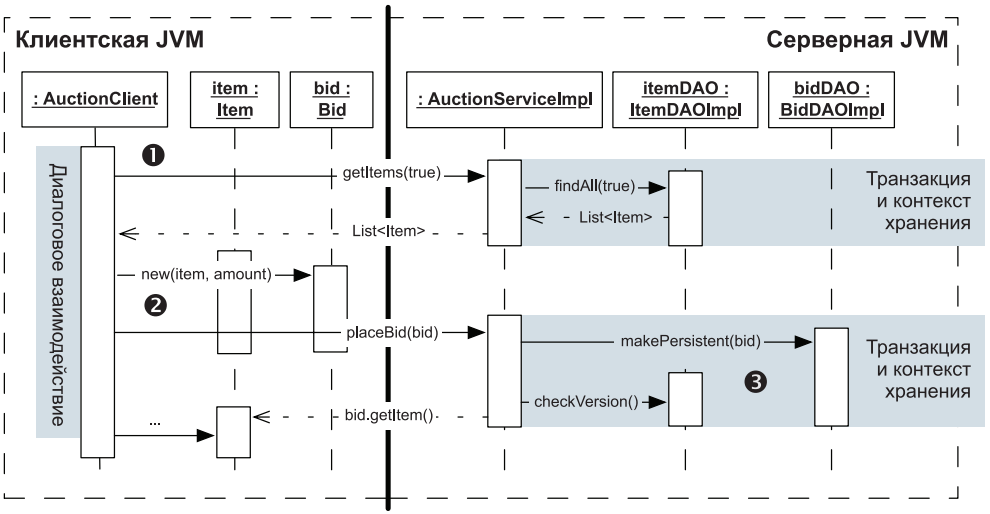


Рис. 18.5 ❖ Вызовы при диалоговом взаимодействии по размещению ставки

И вновь мы пройдемся по коду тестов клиента и сервера. Сначала ❶ клиент получает список экземпляров `Item` и немедленно извлекает коллекции `Item#bids`. Вы видели код этого этапа в предыдущем разделе.

Затем, после получения значения ставки от пользователя ❷, клиент создает новый экземпляр `Bid`, связывая временный экземпляр `Bid` с выбранным отсоединенным экземпляром `Item`. Клиент должен сохранить новый экземпляр `Bid` и отправить его на сервер. Если неправильно документировать API службы, то клиент может попытаться отправить ему отсоединенный экземпляр `Item`:

Файл: /apps/app-stateless-server/src/test/java/org/jpwh/test/stateless/AuctionServiceTest.java

```
item.getBids().add(newBid);

item = service.storeItem(item);
```

«Возможно, эта служба выполняет касакдное сохранение Item#bids, согласно настройкам в аннотации @OneToMany?» После этого ничего не происходит – ставка не сохранилась

В данном случае клиент предположил, что серверу известно о новом объекте `Bid` в коллекции `Item#bids` и что он должен быть сохранен. Ваш сервер может поддерживать подобную функциональность, возможно, за счет включения каскадного слияния в аннотации `@OneToMany` перед этой коллекцией. В таком случае метод `storeItem()` вашей службы будет работать как в предыдущем разделе, посылая экземпляр `Item` интерфейсу `ItemDAO` для сохранения его и его транзитивных зависимостей.

Но наше приложение реализовано иначе: служба предоставляет дополнительный метод `placeBid()`. Перед сохранением ставки в базу данных нужно сделать дополнительную проверку – например, убедиться, что ее значение больше самой большой текущей ставки. Вам также потребуется увеличить номер версии экземпляра `Item`, чтобы избежать добавления конкурирующих ставок. Следовательно, каскадное поведение нужно документировать в связях сущностей предметной модели: коллекция `Item#bids` не является транзитивной, и новые экземпляры `Bid` должны сохраняться только с помощью метода `placeBid()` службы.

Реализация серверного метода `placeBid()` выполняет проверку содержимого и версии:

Файл: /apps/app-stateless-server/src/main/java/org/jpwh/stateless/AuctionServiceImpl.java

```
public class AuctionServiceImpl implements AuctionService {

    // ...

    @Override
    public Item placeBid(Bid bid) throws InvalidBidException {
        bid = bidDAO.makePersistent(bid);

        if (!bid.getItem().isValidBid(bid))  ← Проверка соответствия бизнес-правилам
            throw new InvalidBidException("Bid amount too low!");

        itemDAO.checkVersion(bid.getItem(), true);

        return bid.getItem();
    }
}
```

Здесь можно заметить две интересные особенности. Во-первых, транзакция активна в течение всего вызова `placeBid()`. Метод вложенного компонента EJB вызывает `ItemDAO` и `BidDAO` в той же транзакции. То же самое верно и в отношении контекста хранения: его область действия совпадает с областью действия транзакции ❸. Оба класса DAO объявляют о необходимости внедрения текущего экземпляра `@PersistenceContext`; контейнер среды выполнения предоставит нужный контекст хранения, связанный с текущей транзакцией. Транзакция и создание контекста хранения, а также их связывание с компонентами EJB без состояния реализуются довольно прямолинейно – они всегда привязываются к вызову.

Во-вторых, бизнес-логика проверки нового экземпляра `Bid` находится в классах предметной модели. Служба вызывает `Item#isValid(Bid)`, передавая ответственность за проверку классу предметной модели `Item`. Вот как она реализована в классе `Item`:

Файл: /apps/app-model/src/main/java/org/jpwh/model/Item.java

```
@Entity
public class Item implements Serializable {

    // ...
    public boolean isValidBid(Bid newBid) {
        Bid highestBid = getHighestBid();
        if (newBid == null)
            return false;
        if (newBid.getAmount().compareTo(new BigDecimal("0")) != 1)
            return false;
        if (highestBid == null)
            return true;
        if (newBid.getAmount().compareTo(highestBid.getAmount()) == 1)
            return true;
            return false;
    }

    public Bid getHighestBid() {
        return getBids().size() > 0
            ? getBidsHighestFirst().get(0) : null;
    }

    public List<Bid> getBidsHighestFirst() {
        List<Bid> list = new ArrayList<>(getBids());
        Collections.sort(list);
        return list;
    }

    // ...
}
```

Метод `isValid()` выполняет несколько проверок, чтобы выяснить, превосходит ли значение текущего объекта `Bid` последнюю ставку. Если однажды понадобится реализовать в аукционной системе стратегию «побеждает наименьшая ставка»,

вам достаточно будет поменять реализацию класса `Item` в предметной модели; для служб и интерфейсов DAO, использующих класс, никакой разницы не будет. (Очевидно, вам понадобится вывести другое сообщение для исключения `InvalidBidException`.)

Сомнения вызывает лишь эффективность метода `getHighestBid()`. Он загружает всю коллекцию `bids` в память, сортирует ее, а затем выбирает один экземпляр `Bid`. Улучшенный вариант мог бы выглядеть так:

Файл: /apps/app-model/src/main/java/org/jpwh/model/Item.java

```
@Entity
public class Item implements Serializable {

    // ...

    public boolean isValidBid(Bid newBid,
                              Bid currentHighestBid,
                              Bid currentLowestBid) {
        // ...
    }

}
```

Служба (или, если угодно, контроллер) по-прежнему ничего не знает о бизнес-логике; ей не нужно знать, должна ли новая ставка быть больше или меньше предыдущей. Реализация службы должна передать значения наибольшей и наименьшей ставок, `currentHighestBid` и `currentLowestBid`, при вызове `Item#isValid()`. Как раз на это мы и намекали ранее, когда говорили, что может понадобиться добавить операции в класс `BidDAO`. Чтобы получить эти ставки самым эффективным способом, без загрузки всей коллекции в память и последующей сортировки, можно использовать запросы.

Теперь приложение готово. Оно реализует два запланированных варианта использования. Давайте сделаем шаг назад и проанализируем результаты.

18.2.3. Анализ приложения без состояния

Мы реализовали диалоговые взаимодействия, каждое из которых с точки зрения пользователя представляет единицу работы. Пользователь ожидает выполнить ряд шагов, изменения в которых будут временными лишь до тех пор, пока не произойдет их подтверждение на этапе, завершающем диалоговое взаимодействие. Обычно последний шаг – это завершающий запрос, посылаемый клиентом серверу. Это очень похоже на описание транзакции, но вам, возможно, придется создать несколько системных транзакций на сервере для завершения конкретного диалогового взаимодействия. Проблема заключается в том, как добиться атомарности для нескольких запросов и системных транзакций.

Диалоговые взаимодействия могут иметь любую продолжительность и сложность. В процессе диалогового взаимодействия отсоединенные данные могут быть загружены более чем одним клиентским запросом. Поскольку отсоединенные сущности на стороне клиента находятся под вашим контролем, вы легко можете сде-

лать диалоговое взаимодействие атомарным, если не будете выполнять слияния, сохранения или удаления на сервере, пока не получите завершающего запроса. Вам решать, как накапливать список изменений и где хранить отсоединенные данные, пока пользователь принимает решение. Просто не вызывайте со стороны клиента никаких служебных операций, сохраняющих изменения на сервере, пока не будете уверены, что готовы «подтвердить» (commit) диалоговое взаимодействие.

Одним из важных вопросов, требующих внимания, является сравнение отсоединенных экземпляров: например, если понадобится загрузить несколько экземпляров `Item` и поместить их во множество `Set` или использовать в качестве ключей словаря `Map`. Поскольку экземпляры будут сравниваться вне области гарантированной идентичности объектов – контекста хранения, – необходимо переопределить методы `equals()` и `hashCode()` класса сущности `Item`, как было показано в разделе 10.3.1. В простейшем диалоговом взаимодействии, где использовался только список отсоединенных экземпляров `Item`, это было не нужно. Мы не сравнивали экземпляров во множестве `Set`, не использовали в качестве ключей в словаре `HashMap`, не проверяли их явно на равенство.

Вы должны использовать версионирование сущности `Item` для работы в многопользовательском приложении, как объяснялось в разделе «Включаем версионирование» в главе 11. При слиянии изменений в методе `AuctionService#storeItem()` Hibernate автоматически увеличит версию экземпляра `Item` (только если экземпляр `Item` был изменен). Следовательно, если несколько пользователей одновременно изменят название товара `Item`, Hibernate возбудит исключение во время подтверждения системной транзакции и выталкивания контекста хранения. При выборе оптимистичной стратегии побеждает пользователь, который первым подтвердит изменения, сделанные в ходе диалогового взаимодействия. Второй пользователь должен увидеть обычное сообщение: «Извините, кто-то уже изменил эти данные; пожалуйста, начните сначала».

Мы только что реализовали систему с *толстым клиентом* (rich client); толстый клиент – это не просто терминал ввода/вывода, а приложение со своим внутренним состоянием, независимым от сервера (вспомните, что сервер не хранит никакого состояния). Одним из преимуществ такого сервера без состояния является возможность обработки запроса пользователя любым сервером. Если на сервере произойдет сбой, можно перенаправить запрос на другой сервер, и диалоговое взаимодействие продолжится. У серверов в кластере нет *ничего общего*; вы можете с легкостью масштабировать систему горизонтально, подключая больше серверов. Очевидно, что все серверы приложений обращаются к общей базе данных, но вам придется побеспокоиться о масштабировании только одного уровня серверов.

Сохранение изменений после выхода из состояния гонки

Во время приемочного тестирования может обнаружиться, что пользователям не нравится начинать диалоговое взаимодействие заново, когда обнаруживается состояние гонки (race condition). Они могут потребовать пессимистической блокировки: чтобы во время редактирования данных товара пользователем А пользова-

тель B не мог увидеть этого товара в диалоге редактирования. Главная проблема не в оптимистической проверке версий по окончании диалогового взаимодействия; проблема в том, что все изменения будут потеряны при запуске нового диалогового взаимодействия.
Вместо простого вывода сообщения об ошибке при попытке одновременного изменения данных можно создать диалог, позволяющий пользователю сохранить ставшие недействительными изменения, вручную выполнить слияние с изменениями, сделанными другим пользователем, а затем сохранить итоговый результат. Но предупреждаем, что реализация такой функциональности может потребовать большого количества времени и Hibernate не сильно вам в этом поможет.

Недостатком такого подхода является необходимость разработки толстого клиента, решения проблем сетевого взаимодействия и сериализации данных. Сложность реализации переносится со стороны сервера на клиента, и вы должны оптимизировать связь клиента с сервером.

Если вместо EJB-клиента вы разрабатываете клиента на JavaScript, который должен работать в нескольких браузерах или использоваться как обычное приложение в различных (мобильных) операционных системах, сделать это может быть очень трудно. Мы советуем использовать такую архитектуру, когда толстый клиент работает в популярных браузерах, где пользователи будут загружать самую последнюю версию клиентского приложения каждый раз, когда они заходят на сайт. Развертывание обычных приложений на нескольких платформах, их сопровождение и обновление могут быть серьезным бременем даже в корпоративных сетях среднего размера, где есть возможность управлять пользовательским окружением.

Работая вне среды EJB, вы должны будете реализовать сериализацию и передачу отсоединенных сущностей между клиентом и сервером. Можете ли вы настроить сериализацию и десериализацию экземпляра `Item`? Что произойдет, если клиент будет написан не на Java? Мы рассмотрим этот вопрос в разделе 19.4.

Далее мы повторно реализуем тот же сценарий, но с применением совершенно иной стратегии. Теперь сервер будет хранить состояние диалога с приложением, а клиентом будет простое устройство ввода/вывода. Это – архитектура с *тонким клиентом* и сервером, хранящим состояние (stateful server).

18.3. Разработка сервера с сохранением состояния

Следующее наше приложение останется таким же простым. Оно будет поддерживать те же варианты использования, что и прежде: редактирование товара и размещение ставки. Пользователи приложения не заметят разницы; консольный EJB-клиент будет по-прежнему выглядеть, как на рис. 18.2 и 18.4.

Сервер будет отвечать за преобразование данных для отображения в формат, понятный тонкому клиенту, например в страницу HTML, отображаемую браузе-

ром. Клиент будет передавать данные пользовательского ввода напрямую серверу, например при отправке формы HTML. Сервер должен расшифровать и преобразовать полученные данные в формат, более пригодный для использования в высокоуровневых операциях предметной модели. Однако мы упростим эту часть и используем удаленные вызовы методов в EJB-клиенте.

Сервер также будет запоминать состояние диалогового взаимодействия, сохраняя его в объекте сеанса, связанном с конкретным клиентом. Обратите внимание, что сеанс существует дольше одного диалогового взаимодействия; пользователь может участвовать в нескольких диалоговых взаимодействиях в течение сеанса. Но если пользователь закроет клиентское приложение, не завершив диалогового взаимодействия, данные этого диалога должны быть в какой-то момент очищены. Для решения этой проблемы сервер обычно использует время ожидания; например, сервер может удалить пользовательский сеанс со всеми данными после определенного периода бездействия. Эта работа как раз подходит для сеансовых компонентов EJB с сохранением состояния – они идеально подходят для реализации данной архитектуры.

Запомнив все эти фундаментальные особенности, реализуем первый вариант использования: редактирование товара.

18.3.1. Редактирование информации о товаре

Новый клиент все так же выводит список товаров, а пользователь выбирает один из них. Это простейшая часть приложения, и серверу не нужно хранить никаких данных о состоянии диалога. Взгляните на последовательность вызовов на рис. 18.6.

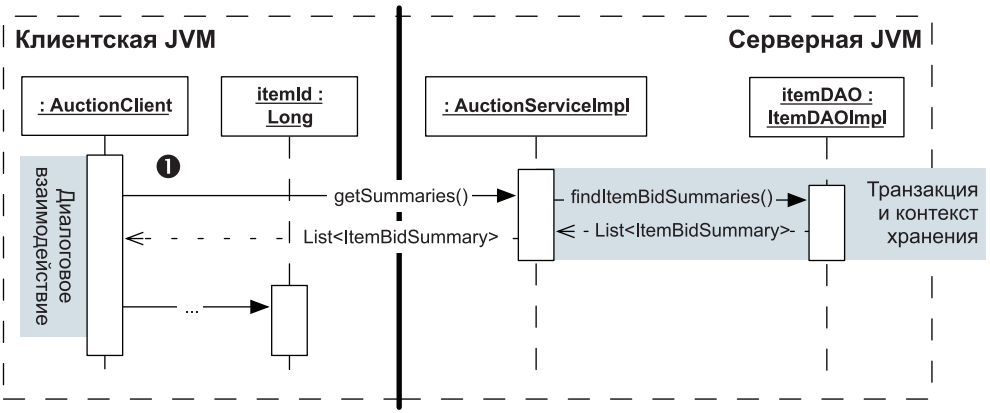


Рис. 18.6 ❖ Клиент получает данные, готовые для отображения

Поскольку клиент очень простой, он не должен знать ничего о классе сущности `Item`. Он загружает ❶ список `List` объектов передачи данных `ItemBidSummary`:

Файл: /apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/AuctionServiceTest.java

```
List<ItemBidSummary> itemBidSummaries = auctionService.getSummaries();
```

Сервер реализует эту функциональность с помощью компонента без состояния, поскольку пока нет необходимости сохранять состояние диалога:

Файл: /apps/app-stateful-server/src/main/java/org/jpwh/stateful/AuctionServiceImpl.java

```
@javax.ejb.Stateless
@javax.ejb.Local(AuctionService.class)
@javax.ejb.Remote(RemoteAuctionService.class)
public class AuctionServiceImpl implements AuctionService {

    @Inject
    protected ItemDAO itemDAO;

    @Override
    public List<ItemBidSummary> getSummaries() {
        return itemDAO.findItemBidSummaries();
    }
}
```

Даже в серверной архитектуре с сохранением состояния всегда будет происходить множество коротких диалоговых взаимодействий с приложением, не требующим сохранения состояния на сервере. Это нормально, и важно понимать, что хранение состояния на сервере стоит ресурсов. Если реализовать операцию `getSummaries()`, используя сеансовый компонент с сохранением состояния, вы лишь впустую потратите ресурсы. Компонент с сохранением состояния понадобится только для единственной операции, после чего он будет занимать память, пока контейнер не избавится от него. Архитектура сервера с состоянием не обязывает применять одни лишь компоненты с состоянием.

Далее клиентское приложение выводит список объектов `ItemBidSummary`, содержащий только идентификатор, описание и максимальную ставку каждого товара. Это именно то, что пользователь видит на экране, как показано на рис. 18.2. После чего пользователь введет идентификатор товара и начнет диалоговое взаимодействие. Схема этого диалогового взаимодействия приводится на рис. 18.7.

Клиент сообщает серверу, что тот должен начать диалоговое взаимодействие, посылая ему значение идентификатора ❷:

Файл: /apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/AuctionServiceTest.java

```
itemService.startConversation(itemId);
```

Здесь уже не вызывается служба `AuctionService` без состояния из предыдущего раздела. Новая служба `ItemService` – это компонент с сохранением состояния; сервер будет создавать его экземпляры и назначать их отдельно для каждого клиента. Эта служба реализуется с использованием сеансового компонента с состоянием:

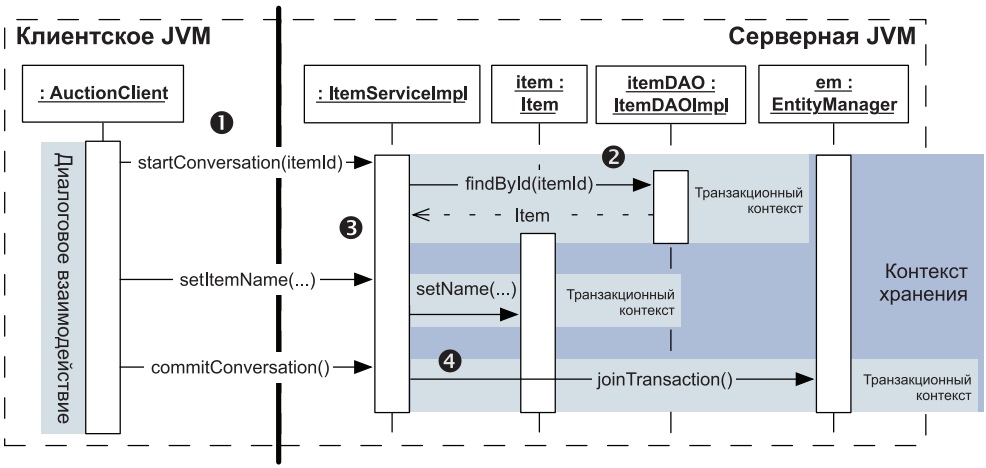


Рис. 18.7 ❖ Клиентское приложение задает границы диалогового взаимодействия на сервере

Файл: /apps/app-stateful-server/src/main/java/org/jpwh/stateful/ItemServiceImpl.java

```
@javax.ejb.Stateful(passivationCapable = false)
@javax.ejb.StatefulTimeout(10)  ← Минуты
@javax.ejb.Local(ItemService.class)
@javax.ejb.Remote(RemoteItemService.class)
public class ItemServiceImpl implements ItemService {

    @PersistenceContext(type = EXTENDED, synchronization = UNSYNCHRONIZED)
    protected EntityManager em;

    @Inject
    protected ItemDAO itemDAO;

    @Inject
    protected BidDAO bidDAO;

    // Состояние диалогового взаимодействия на сервере
    protected Item item;
    // ...

    @Override
    public void startConversation(Long itemId) {
        item = itemDAO.findById(itemId);
        if (item == null)
            throw new EntityNotFoundException(
                "No Item found with identifier: " + itemId
            );
    }

    // ...
}
```

Этот класс имеет множество аннотаций, определяющих взаимодействие контейнера с компонентом. Поскольку для компонента задано время ожидания 10 минут, сервер удалит и уничтожит его, если он не будет вызываться в течение этого времени. Это позволяет избавиться от диалоговых взаимодействий с длительным периодом бездействия, когда, например, пользователь долго не использует клиентского приложения.

Также для компонента EJB отключено пассивирование (passivation): контейнер EJB может сериализовать и сохранить компонент с состоянием на диск для экономии памяти или передачи его другому узлу кластера для восстановления сеанса в случае ошибки. Пассивирование не затронет лишь одного поля – `EntityManager`. Контекст хранения присоединяется к этому компоненту благодаря параметру `EXTENDED`; также класс `EntityManager` не реализует интерфейс `java.io.Serializable`.

ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ:
Почему нельзя сериализовать EntityManager?

Нет никаких технических ограничений, почему контекст хранения и объект `EntityManager` не могут быть сериализованы. Конечно, после десериализации объект `EntityManager` должен быть присоединен к правильному экземпляру `EntityManagerFactory` на целевой машине, но это уже особенности реализации. Пассивирование контекста хранения до сих пор было вне поля зрения спецификаций JPA и Java EE. Тем не менее большинство реализаций позволяет сериализовать и корректно десериализовать контекст хранения. `EntityManager` в Hibernate может быть сериализован и десериализован и правильно присоединен к нужной единице хранения после десериализации.

Используя Hibernate с сервером Wildfly, вы *могли бы* использовать пассивирование в предыдущем примере, получая возможность восстановления сеанса в случае ошибки, а также бесперебойную работу сервера с состоянием и расширенные контексты хранения. Тем не менее эта функциональность не стандартизована; как мы увидим далее, такая стратегия препятствует масштабированию.

Были времена, кода даже в Hibernate не было возможности сериализовать `EntityManager`. Вы можете столкнуться с устаревшими фреймворками, пытавшимися обойти это ограничение, такими как Seam, который использует `ManagedEntityInterceptor`. Вам следует избегать этого и находить более простые решения, такие как перенаправление запросов на один узел кластера (sticky session), архитектуру сервера с состоянием или внедрение зависимостей (CDI) при диалоговом взаимодействии на стороне сервера, которое мы обсудим в следующей главе на примере контекста хранения, связанного с запросом.

Аннотация `@PersistenceContext` объявляет, что этот компонент с состоянием нуждается в экземпляре `EntityManager` и контейнер должен *расширять* время жизни контекста хранения до границ жизненного цикла компонента. Режим расширения доступен только для EJB-компонентов с состоянием. Без этого контейнер будет создавать и закрывать контекст хранения при подтверждении каждой транзакции. Но здесь требуется, чтобы контекст хранения оставался

открытым за границами транзакции и был привязан к экземпляру сеансового компонента с состоянием.

Более того, нужно предотвратить автоматическое выталкивание контекста во время подтверждения транзакции, поэтому используется параметр `UNSYNCHRONIZED`. Hibernate вытолкнет контекст хранения после его подключения к транзакции вручную. Теперь Hibernate не будет автоматически записывать изменения в хранимых экземплярах сущностей в базу данных; вместо этого он будет накапливать изменения, пока вы не решите записать все разом.

В начале диалогового взаимодействия сервер загрузит экземпляр `Item`, представляющий состояние диалога, и сохранит его в поле класса ❷ (рис. 18.7). Объекту `ItemDAO` также необходим экземпляр `EntityManager`; вспомните, что он отмечен аннотацией `@PersistenceContext` без дополнительных параметров. Правила передачи контекста хранения в EJB такие же, как и раньше: Hibernate передает контекст хранения вместе с контекстом транзакции. Контекст хранения будет передан в `ItemDAO` вместе с транзакцией, запущенной вызовом метода `startConversation()`. После выхода из метода `startConversation()` транзакция подтверждается, но контекст хранения не выталкивается и не закрывается. Экземпляр `ItemServiceImpl` ожидает следующего обращения клиента.

Следующий вызов от клиента просит сервер изменить название товара ❸:

Файл: /apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/AuctionServiceTest.java

```
itemService.setItemName("Pretty Baseball Glove");
```

На стороне сервера транзакция запустится вызовом метода `setItemName()`. Но поскольку никакие ресурсы транзакции не используются (ни вызовы DAO, ни вызовы `EntityManager`), изменится только объект `Item`, представляющий диалоговое состояние:

Файл: /apps/app-stateful-server/src/main/java/org/jpwh/stateful/ItemServiceImpl.java

```
public class ItemServiceImpl implements ItemService {

    // ...
    @Override
    public void setItemName(String newName) {
        item.setName(newName);
    }          ← После подтверждения транзакции контекст хранения не выталкивается,
                 поскольку он рассинхронизирован
    // ...
}
```

Обратите внимание, что экземпляр `Item` по-прежнему находится в хранимом состоянии, потому что контекст хранения еще открыт! Но из-за рассинхронизации он не обнаружит изменений в экземпляре `Item`, поскольку не будет выталкиваться при подтверждении транзакции.

Наконец, клиентское приложение завершает диалоговое взаимодействие, предлагая серверу сохранить изменения (рис. 18.7):

Файл: /apps/app-stateful-server/src/test/java/org/jpwh/test/stateful/AuctionServiceTest.java

```
itemService.commitConversation();
```

Теперь на сервере можно записать изменения в базу данных и очистить диалоговое состояние:

Файл: /apps/app-stateful-server/src/main/java/org/jpwh/stateful/ItemServiceImpl.java

```
public class ItemServiceImpl implements ItemService {

    @Override
    @javax.ejb.Remove  <-- Компонент удаляется после завершения этого метода
    public void commitConversation() {
        em.joinTransaction();
    }  <-- Контекст хранения соединяется с текущей транзакцией
           и выталкивается после выхода из метода, сохраняя изменения
}
```

На этом реализация первого варианта использования закончена. Мы опускаем реализацию второго варианта (размещение ставки) и отсылаем вас к коду примера за подробностями. Код второго варианта очень похож на код первого, поэтому у вас не должно возникнуть проблем с его пониманием. Важно, чтобы вы понимали, как работают контекст хранения и как действуют транзакции в EJB.

> **ПРИМЕЧАНИЕ** В EJB существуют дополнительные правила передачи контекста хранения между различными типами компонентов. Они довольно сложны, и мы никогда не видели хороших вариантов для их применения. К примеру, вряд ли вы станете вызывать компонент EJB с состоянием из компонента EJB без состояния. Еще одну трудность представляют методы EJB с отключенными или необязательными транзакциями, которые также влияют на передачу контекста хранения через вызовы компонентов. Мы рассказывали об этих правилах в предыдущем издании этой книги. Советуем придерживаться только тех стратегий, что были показаны в этой главе, ничего не усложняя.

Давайте обсудим некоторые различия между архитектурами с состоянием и без состояния.

18.3.2. Анализ приложений с сохранением состояния

Так же, как при анализе приложения без состояния, рассмотрим сначала реализацию единицы работы с точки зрения пользователя. В частности, нужно понять, как в диалоговом взаимодействии реализована атомарность и как представить последовательность действий в виде одной единицы работы.

В какой-то момент – обычно во время последнего запроса в рамках диалогового взаимодействия – происходят подтверждение и запись изменений в базу данных.

Диалоговое взаимодействие будет атомарным, если не присоединять расширенного экземпляра `EntityManager` с транзакцией до последнего диалогового события. Если читать данные в рассинхронизированном режиме, состояние объектов проверяться не будет, так же как не будет выталкиваться контекст хранения.

Пока контекст открыт, можно выполнять отложенную загрузку данных, обращаясь к прокси-объектам и незагруженным коллекциям, что, очевидно, довольно удобно. Загруженный объект `Item`, как и прочие данные, устаревает, если пользователю требуется длительное время для выполнения следующего запроса. Во время диалогового взаимодействия вам, возможно, придется выполнять операцию `refresh()` для некоторых управляемых экземпляров сущностей, чтобы получать обновления из базы данных, как объяснялось в разделе 10.2.6. С другой стороны, вы можете выполнять обновления для отката изменений, сделанных в ходе диалогового взаимодействия. Например, если пользователь поменяет поле `Item#name`, а затем решит отменить изменение, вы можете обновить хранимый экземпляр `Item` вызовом метода `refresh()`, который извлечет из базы данных старое название товара. Эта приятная особенность расширенного контекста хранения позволяет экземпляру `Item` всегда находиться в управляемом состоянии.

Точки сохранения в диалоговых взаимодействиях

Вам могут быть знакомы точки сохранения (savepoints) в транзакциях JDBC: после изменения некоторых данных в рамках транзакции создается точка сохранения; позже вы можете откатить транзакцию до этой точки, отказываясь лишь от части сделанных изменений, но сохраняя все, что было сделано до создания точки восстановления. К сожалению, Hibernate не поддерживает ничего, похожего на точки восстановления, для контекста хранения. Экземпляр сущности можно откатить только до состояния в базе данных, используя метод `refresh()`. В Hibernate можно использовать обычные точки сохранения в транзакциях JDBC (для этого потребуется экземпляр `Connection`; см. раздел 17.1), но они не помогут сделать откат в диалоговом взаимодействии.

Серверная архитектура с сохранением состояния труднее поддается горизонтальному масштабированию. Если на сервере произойдет сбой, текущее состояние диалогового взаимодействия, как и сеанс, целиком будет потеряно. Репликация сеанса на несколько серверов – это дорогостоящая операция, поскольку каждое изменение в сеансе на одном сервере вызывает сетевое взаимодействие с другими (потенциально всеми) серверами.

Сериализация расширенного контекста хранения невозможна при работе с компонентами EJB и расширенными экземплярами `EntityManager`. При использовании компонентов EJB с состоянием и расширенного контекста хранения в кластере можно рассмотреть вариант применения *прикрепленного сеанса* (sticky session), когда запросы конкретного клиента всегда направляются на один физический сервер. Это позволит справиться с растущей нагрузкой путем добавления серверов, но пользователь должен быть готов к потере данных в случае сбоя на сервере.

С другой стороны, сервер с состоянием может выступать в качестве первой линии кэширования со своими расширенными контекстами хранения в сеансах пользователей. Как только экземпляр `Item` будет загружен во время диалогового взаимодействия с конкретным пользователем, он не будет загружаться снова из базы данных в рамках этого взаимодействия. Это может стать отличным инструментом для снижения нагрузки на сервер базы данных (самый дорогой слой с точки зрения масштабирования).

Стратегия применения расширенного контекста хранения требует от сервера больше памяти, чем хранение только отсоединенных экземпляров, потому что контекст хранения в Hibernate содержит копии всех управляемых экземпляров. Вам может потребоваться вручную отсоединять управляемые экземпляры методом `detach()` для управления тем, что хранится в контексте, или отключать проверку состояния объектов и хранение копий (сохраняя при этом возможность отложенной загрузки), как объяснялось в разделе 10.2.8.

Конечно, существуют альтернативные реализации тонких клиентов и серверов с состоянием. Можно использовать контекст хранения, связанный с запросом, управляя отсоединенными (нехранимыми) экземплярами сущностей на сервере вручную. Очевидно, это можно сделать путем отсоединения и слияния, но потребует больших затрат на реализацию. Одно из главных преимуществ расширенного контекста хранения – прозрачная отложенная загрузка (даже между запросами) – больше не будет доступна. В следующей главе мы покажем реализацию такого сервера с состоянием, контекст хранения которого в CDI и JSF привязан к запросу, и вы сможете сравнить это с функциональностью расширенного контекста хранения EJB, которую мы показали в этой главе.

Системы с тонким клиентом, как правило, создают большую нагрузку на сервер, чем толстые клиенты. Каждый раз, когда пользователь взаимодействует с приложением, любое клиентское событие будет отправлять запрос по сети. Это может происходить при каждом нажатии на кнопку мыши в веб-приложении. Только сервер знает о состоянии диалогового взаимодействия, и он должен подготовить и отобразить информацию, которую увидит пользователь. Толстый клиент, напротив, может загрузить необработанные данные в одном запросе, преобразовать их и уже на месте привязать их к пользовательскому интерфейсу. Диалог в толстом клиенте может накапливать изменения на стороне клиента и отправлять запрос по сети только в конце взаимодействия, когда нужно сохранить изменения в базу данных.

Другая проблема с тонким клиентом заключается в параллельных диалоговых взаимодействиях одного пользователя: что произойдет, если пользователь изменит два элемента одновременно, например в двух вкладках браузера? Это будет означать, что пользователь запустит два диалоговых взаимодействия на сервере. Сервер должен будет разделить данные в сеансе пользователя, в зависимости от диалогового взаимодействия. Следовательно, клиентские запросы должны будут содержать некоторое подобие идентификатора диалогового взаимодействия, чтобы можно было извлекать корректное состояние диалога из сеанса пользовате-

ля при каждом запросе. При работе с клиентами и серверами, основанными на EJB, это происходит автоматически, но вряд ли эта функциональность встроена в ваш любимый фреймворк веб-приложений (если только это не JSF и CDI, как вы узнаете в следующей главе).

Одним из самых больших преимуществ сервера с состоянием является слабая зависимость от клиентской платформы; если клиентом является простой терминал ввода/вывода, будет меньше шансов, что что-то пойдет не так. Единственное место проверки данных и безопасности будет на сервере. Не будет никаких проблем с развертыванием; вы сможете обновлять приложение на сервере, не затрагивая клиентов.

Сегодня у тонкого клиента немного преимуществ, и количество установок серверов с состоянием снижается. Это особенно заметно на рынке веб-приложений, где легкость масштабирования является решающим фактором.

18.4. Резюме

- В этой главе мы реализовали простые диалоговые взаимодействия – единицы работы с точки зрения пользователей приложения.
- Рассмотрели проекты клиента и сервера с состоянием и без состояния и узнали, как Hibernate вписывается в каждую из них.
- Вы можете работать либо с отсоединенными экземплярами сущностей, либо с расширенным контекстом хранения, распространяющимся на все диалоговое взаимодействие.

Глава 19

Создание веб-приложений

В этой главе:
- интеграция JPA с CDI и JSF;
- просмотр таблиц баз данных;
- реализация длительных диалоговых взаимодействий;
- настройка сериализации сущностей.

В этой главе вы узнаете, как применять Hibernate в типичном веб-приложении. Существуют десятки фреймворков для разработки веб-приложений на Java, поэтому мы заранее приносим извинения, если пройдем мимо вашего любимого фреймворка. Мы рассмотрим применение JPA в стандартном окружении Java Enterprise Edition, в частности в сочетании со следующими стандартами: внедрение контекста и зависимостей (CDI), Java Server Faces (JSF) и Java API для веб-служб на основе REST (JAX-RS). Мы, как всегда, продемонстрируем шаблоны, которые можно применять также в нестандартных окружениях.

Сначала мы еще раз обратимся к уровню хранения и продемонстрируем применение CDI в классах DAO. После чего расширим эти классы, реализовав обобщенное решение для сортировки и постраничного вывода данных. Это решение можно использовать, когда потребуется отобразить табличные данные независимо от выбранного фреймворка.

Далее, опираясь на уровень хранения, мы напишем полностью работающее JSF-приложение и познакомимся с областью видимости диалога в Java EE (*conversation scope*), в которой CDI, взаимодействуя с JSF, поддерживает простую модель серверных компонентов с состоянием. Если вам не приглянулись показанные в прошлой главе EJB-компоненты с состоянием и расширенным контекстом хранения, диалоговые взаимодействия с отсоединенным состоянием сущности на сервере, вероятно, вам понравятся больше.

И наконец, если вам нравится создавать веб-приложения с толстыми клиентами, серверами без состояния и такими фреймворками, как JAX-RS, GWT или AngularJS, мы научим вас сериализовать экземпляры сущностей JPA в форматы XML и JSON. А начнем мы с переноса реализации уровня хранения с EJB на CDI.

19.1. Интеграция JPA и CDI

Стандарт CDI определяет механизм типизированного внедрения зависимостей, а также систему управления жизненным циклом компонентов в среде выполнения Java EE. В предыдущей главе вы уже использовали аннотацию `@Inject` для связывания компонентов `ItemDAO` и `BidDAO` с классами служб EJB.

JPA-аннотация `@PersistenceContext`, которую мы помещали *внутрь* классов DAO, представляет еще один, особый случай внедрения зависимостей: вы просите контейнер среды выполнения внедрить экземпляр `EntityManager` и автоматически управлять им. Этот экземпляр, *EntityManager*, будет `управляться контейнером`. Но есть еще подводные камни, такие как передача контекста хранения и правила распространения транзакций, которые мы обсудили в предыдущей главе. Такие правила удобны, когда все классы служб и DAO являются компонентами EJB; но если вы не используете EJB, вам, возможно, не захочется следовать этим правилам. С помощью *управляемого приложением* экземпляра `EntityManager` можно определить собственные правила управления контекстом хранения, его передачи и внедрения.

Сейчас мы перепишем классы DAO, как обычные управляемые компоненты CDI, которые очень похожи на EJB: простые Java-классы с дополнительными аннотациями. Нужно лишь с помощью аннотации `@Inject` внедрить экземпляр `EntityManager`, избавившись от `@PersistenceContext`, и получить тем самым полный контроль над контекстом хранения. Но прежде, чем внедрить собственный экземпляр `EntityManager`, его нужно *создать*.

19.1.1. Создание экземпляра EntityManager

Продюсер (*producer*) в терминологии CDI – это фабрика, управляющая созданием экземпляра, которую контейнер среды выполнения будет вызывать, когда приложению понадобится экземпляр в заданной области видимости. К примеру, контейнер создаст экземпляр в области видимости приложения только один раз за весь жизненный цикл приложения. Экземпляр в области видимости запроса (request-scoped) будет создаваться контейнером при получении запроса от клиента, а экземпляр в области видимости сеанса (session-scoped) – отдельно для каждого сеанса пользователя.

Спецификация CDI задает отображение между абстрактными понятиями *запроса* и *сеанса* и реальными объектами запроса и сеанса в сервлете. Не забывайте, что JSF и JAX-RS построены на основе сервлетов, поэтому CDI отлично подходит для этих фреймворков. Другими словами, не беспокойтесь об этом: в окружении Java EE вся работа по интеграции уже сделана за вас.

Давайте напишем продюсера для экземпляров `EntityManager` в области видимости запроса:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/EntityManagerProducer.java

```
@javax.enterprise.context.ApplicationScoped  ← ❶ Нужен только 1 продюсер
public class EntityManagerProducer {
```

```
    @PersistenceUnit                                    ← ❷ Получение единицы хранения
    private EntityManagerFactory entityManagerFactory;

    @javax.enterprise.inject.Produces                   ← ❸ Получение экземпляра EntityManager
    @javax.enterprise.context.RequestScoped
    public EntityManager create() {
        return entityManagerFactory.createEntityManager();
    }

    public void dispose(
        @javax.enterprise.inject.Disposes               ← ❹ Закрытие контекста хранения
        EntityManager entityManager) {
        if (entityManager.isOpen())
            entityManager.close();
    }
}
```

❶ Эта аннотация CDI объявляет, что во всем приложении должен иметься только один продюсер: будет существовать единственный экземпляр `EntityManagerProducer`.

❷ Среда выполнения Java EE предоставит единицу хранения, определенную в файле persistence.xml, которая также является компонентом области видимости приложения. (Если CDI используется вне окружения Java EE, можно вызвать статический фабричный метод `Persistence.createEntityManagerFactory()`).

❸ Как только приложению потребуется экземпляр `EntityManager`, будет вызван метод `create()`. Контейнер повторно использует один экземпляр `EntityManager` в течение всего запроса, обрабатываемого сервером. (Если забыть поместить аннотацию `@RequestScoped` перед методом, экземпляр `EntityManager` будет иметь область видимости приложения, как и класс продюсера!)

❹ После завершения обработки запроса, при удалении его контекста, контейнер CDI вызовет этот метод, чтобы избавиться от экземпляра `EntityManager`. Поскольку вы сами создали этот управляемый приложением контекст хранения (см. раздел 10.1.2), вы и отвечаете за его закрытие.

Частой ошибкой при работе с классами, использующими аннотации CDI, является неправильный импорт аннотаций. В Java EE 7 существуют две аннотации с именем `@Produces`; вторая находится в пакете `javax.ws.rs` (спецификация JAX-RS). Ее семантика отличается от аннотации продюсера CDI, и вы можете часами искать ошибку, если импортировали не ту аннотацию. Другой такой же аннотацией является `@RequestScoped` из пакета `javax.faces.bean` (спецификация JSF). Как и большинство устаревших аннотаций JSF для управления компонентами из пакета `javax.faces.bean`, ее не стоит использовать, если доступна более современная альтернатива из CDI. Мы надеемся, что будущие спецификации Java EE устранят эту двусмысленность.

Теперь у нас есть фабрика для создания экземпляров `EntityManager`, управляемых приложением, и контекста хранения с областью видимости запроса. Теперь нужно придумать, как сообщать экземпляру `EntityManager` о системных транзакциях.

19.1.2. Присоединение экземпляра EntityManager к транзакциям

Когда серверу потребуется обработать запрос сервлета, контейнер автоматически создаст экземпляр `EntityManager` при первой необходимости его внедрения. Помните, что созданный вручную экземпляр `EntityManager` автоматически присоединится к системной транзакции, только если та уже началась. В противном случае он будет *рассинхронизирован*: вы будете читать данные в режиме автоматического подтверждения (auto-commit), и Hibernate не будет выталкивать контекста хранения.

Не всегда очевидно, в какой момент контейнер вызовет продюсера `EntityManager` или когда точно во время обработки запроса произойдет внедрение `EntityManager`. При обработке запроса вне системной транзакции получаемый объект `EntityManager` всегда будет рассинхронизирован. Следовательно, мы должны сделать так, чтобы экземпляр `EntityManager` узнал о системной транзакции.

Для этой цели в суперинтерфейсе уровня хранения имеется следующий метод:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/GenericDAO.java

```
public interface GenericDAO<T, ID extends Serializable>
    extends Serializable {

    void joinTransaction();

    // ...
}
```

Мы должны вызвать этот метод в каждом классе DAO перед сохранением данных, когда точно известно, что он будет вызван в рамках транзакции. Напомним, что при попытке записи данных Hibernate возбудит исключение `TransactionRequiredException`, напоминая, что экземпляр `EntityManager` был создан перед началом транзакции и не знает о ней. Если вы хотите потренировать свои навыки в CDI, можете попробовать реализовать эту функциональность с помощью *декораторов* или *перехватчиков* CDI.

Давайте реализуем новый метод интерфейса `GenericDAO`, связав экземпляр `EntityManager` с классами DAO.

19.1.3. Внедрение экземпляра EntityManager

Старая реализация `GenericDAOImpl` полагалась на аннотацию `@PersistenceContext` для внедрения объекта `EntityManager` в поле класса или на вызов `setEntityManager()` перед использованием класса DAO. С помощью CDI можно использовать более безопасную технику внедрения в конструктор:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/GenericDAOImpl.java

```
public abstract class GenericDAOImpl<T, ID extends Serializable>
        implements GenericDAO<T, ID> {

    protected final EntityManager em;
    protected final Class<T> entityClass;
```

```
    protected GenericDAOImpl(EntityManager em, Class<T> entityClass) {
        this.em = em;
        this.entityClass = entityClass;
    }

    public EntityManager getEntityManager() {
        return em;
    }

    @Override
    public void joinTransaction() {
        if (!em.isJoinedToTransaction())
            em.joinTransaction();
    }
    // ...
}
```

Каждый, кто захочет создать экземпляр класса DAO, должен будет передать ему объект `EntityManager`. Такое определение инварианта класса дает более весомые гарантии; следовательно, несмотря на то что в наших примерах мы часто используем внедрение в поля классов, вы должны в первую очередь рассматривать возможность внедрения в конструкторы. (Мы не делаем этого в некоторых примерах, поскольку от этого они стали бы только длиннее, а книга уже и так не маленькая).

В конкретных подклассах (DAO сущностей) следует объявить требуемое внедрение в конструктор:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/ItemDAOImpl.java

```
public class ItemDAOImpl
    extends GenericDAOImpl<Item, Long>
    implements ItemDAO {

    @Inject
    public ItemDAOImpl(EntityManager em) {
        super(em, Item.class);
    }

    // ...

}
```

Когда приложению потребуется объект `ItemDAO`, среда выполнения CDI обратится к продюсеру `EntityManagerProducer`, а затем вызовет конструктор `ItemDAOImpl`. В рамках одного запроса контейнер повторно использует один и тот же объект `EntityManager` для внедрения в каждый экземпляр DAO.

ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ: Как в CDI работать с несколькими единицами хранения?

При работе с несколькими базами данных – разными единицами хранения – можно использовать *квалификаторы* CDI, чтобы различать их. Квалификатор – это произ-

вольная аннотация. Вы создаете аннотацию вроде `@BillingDatabase` и отмечаете ее как квалификатор. Затем помещаете ее рядом с аннотацией `@Produces` перед методом, создающим экземпляр `EntityManager` для этой конкретной единицы хранения. Теперь, когда понадобится этот экземпляр `EntityManager`, вы должны будете добавить аннотацию `@BillingDatabase` рядом с `@Inject`.

В какой области видимости находится `ItemDAO`? Поскольку область видимости классов реализации не задана, это *зависит от конкретной ситуации*. Экземпляр `ItemDAO` создается, когда он кому-либо нужен, поэтому экземпляр `ItemDAO` будет находиться в той же области видимости, что и вызывающий код, и будет принадлежать вызывающему объекту. Это хорошее решение для реализации интерфейса уровня хранения, поскольку перекладывает решение о выборе области видимости на верхний уровень, состоящий из служб, обращающихся к уровню хранения.

Теперь с помощью аннотации `@Inject` можно внедрить экземпляр `ItemDAO` в поле класса службы. Но, прежде чем воспользоваться уровнем хранения на основе CDI, давайте реализуем еще поддержку сортировки и постраничной выборки данных.

19.2. Сортировка и постраничная выборка данных

Очень распространенным требованием являются загрузка данных из базы и их отображение на веб-странице в табличном виде. Но часто также требуется реализовать динамическую *постраничную выборку* и *сортировку* данных:

- поскольку запрос возвращает больше данных, чем можно отобразить на одной странице, вы должны показывать только часть из них. Вы отображаете определенное количество записей, давая пользователю возможность перехода к следующему, предыдущему, первому или последнему набору записей. Пользователь также ожидает сохранения порядка сортировки при переключении между страницами;
- пользователь должен иметь возможность, щелкнув мышкой на заголовке колонки, отсортировать строки в таблице по значениям в этой колонке. Обычно сортировать можно либо *по возрастанию*, либо *по убыванию*; направление можно менять последовательными щелчками на заголовке колонки.

Сейчас мы реализуем обобщенное решение для постраничной выборки, основанное на метамодели хранимых классов, предоставляемой JPA.

Постраничную выборку можно реализовать двумя способами: с помощью приема *смещения* (offset) или *поиска* (seek). Давайте рассмотрим их отличия и определимся с реализацией.

19.2.1. Реализация постраничной выборки с помощью смещения или поиска

На рис. 19.1 показан пример пользовательского интерфейса с постраничным отображением на основе смещений. У нас имеется довольно много аукционных товаров, но на одной странице можно отобразить только три записи. Сейчас мы

находимся на первой странице; приложение также динамически отображает ссылки на другие страницы. Результаты отсортированы по возрастанию наименования товара. Вы можете щелкнуть на заголовке колонки, чтобы отсортировать по убыванию (или возрастанию) наименования, даты окончания аукциона или наибольшему значению ставки. Щелчок на наименовании товара в таблице откроет диалог просмотра данных о товаре, в котором вы сможете сделать ставку. Как раз этот вариант использования мы и реализуем в этой главе.

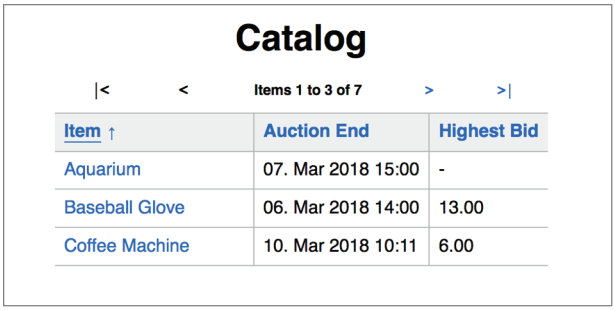


Рис. 19.1 ❖ Отображение страниц каталога с использованием приема смещений

Для создания этой страницы применялись запросы к базе данных с заданными смещением и количеством извлекаемых записей. Для этого вызывались методы Java Persistence API: `Query#setFirstResult()` и `Query#setMaxResults()`, обсуждавшиеся в разделе 14.2.4. Сначала пишется запрос, а затем фреймворку Hibernate предоставляется возможность добавить в него ограничения на смещение и количество извлекаемых записей, в зависимости от используемого диалекта SQL.

Теперь пришло время рассмотреть альтернативный подход с применением поиска, как показано на рис. 19.2. Здесь у пользователя нет возможности перейти на произвольную страницу с заданным смещением; он может листать только вперед, переходя на следующую страницу. Это может выглядеть как ограничение, но вы, вероятно, видели или даже использовали такой способ постраничной выборки, когда требовалась *бесконечная прокрутка*. Можно, например, автоматически подгрузить и отобразить следующую страницу с данными, когда пользователь дойдет до конца таблицы/экрана.

Метод поиска основан на особом ограничении в запросе, извлекающем данные. Когда придет время загрузить следующую страницу, выполнится поиск всех товаров с названиями «больше, чем [Coffee Machine]». Движение вперед будет осуществляться не за счет установки смещения в результатах методом `setFirstResult()`, а за счет ограничения результатов на основе отсортированных значений какого-либо ключа. Если вы незнакомы с постраничной выборкой на основе поиска, иногда называемой *ключевой* (keyset paging), мы уверены, что она не покажется вам сложной после тех запросов, которые вы увидите далее в этой главе.

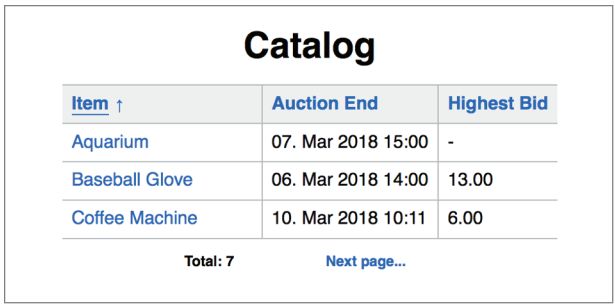


Рис. 19.2 ❖ Отображение страниц каталога с использованием приема поиска следующей страницы

Давайте обсудим недостатки и преимущества обоих подходов. Конечно, можно реализовать бесконечную прокрутку, используя постраничную выборку на основе смещения, или переход к конкретной странице, используя метод поиска; но у каждой из них есть свои сильные и слабые стороны:

- метод смещения удобен, если пользователю нужно перейти непосредственно к конкретной странице. К примеру, большинство поисковых движков поддерживает переход прямо к странице 42 в результатах запроса или сразу к последней странице. Поскольку мы с легкостью можем рассчитать смещение и количество извлекаемых записей в зависимости от требуемого номера страницы, мы без труда сможем это реализовать. Реализовать подобный пользовательский интерфейс с помощью метода поиска гораздо сложнее; необходимо заранее знать искомое значение. Поскольку неизвестно, какой товар предшествует странице 42, мы не сможем выбрать все товары с наименованием «больше, чем X». Метод поиска подходит только для пользовательских интерфейсов, где пользователи переходят в прямом или обратном направлении от страницы к странице в списке или таблице с данными и где известно последнее (или первое) значение, показанное пользователю;
- отличный вариант использования для постраничной выборки на основе поиска основан на ключевых значениях, которые не нужно запоминать. Например, все пользователи с именами, начинающимися на *C*, могут отображаться на одной странице, а пользователи с именами, начинающимися на *D*, – на следующей. Также каждая страница может отображать только товары, преодолевшие пороговое значение максимальной ставки;
- метод смещений работает гораздо хуже для страниц, находящихся в конце. При переходе к странице 5000 база данных должна подсчитать все строки и подготовить 5000 страниц данных, прежде чем сможет пропустить предыдущие 4999. Типичное решение этой проблемы заключается в ограничении количества страниц, на которые пользователь может непосредственно перейти: например, разрешить пользователю переход только на первые 100 страниц, заставляя его уточнить ограничения запроса для получения меньшего количества результатов. Метод поиска обычно работает быстрее

метода смещений даже для самых первых страниц. Оптимизатор запросов базы данных может сразу перейти к началу требуемой страницы и эффективно ограничить сканируемую область индекса. Записи, показанные на предыдущих страницах, никак не учитываются и не подсчитываются;

- иногда метод смещений может показывать некорректные результаты. Хотя результат будет соответствовать состоянию базы данных, пользователям он может казаться некорректным. Когда приложение вставляет или удаляет записи во время просмотра данных пользователем, могут возникать аномалии. Представьте пользователя, который смотрит на страницу 1, в то время как другой добавляет данные, которые должны появиться на странице 1. Если теперь пользователь перейдет к странице 2, некоторые записи, которые он мог видеть на странице 1, перейдут на страницу 2. Если запись со страницы 1 была удалена, пользователь может не увидеть некоторых записей со страницы 2, поскольку они перейдут на страницу 1. В методе поиска таких аномалий не возникает; записи мистически не появляются и не исчезают.

Сейчас мы покажем, как реализовать оба метода постраничной выборки путем расширения уровня хранения. Начнем с простой модели, хранящей номер текущей страницы и настройки сортировки табличных данных.

19.2.2. Реализация постраничной выборки в уровне хранения

Для координации запросов и отображения страниц с данными нужно хранить информацию о размере страницы и знать, какая страница сейчас отображается. Ниже показан простейший класс для хранения этой информации; это абстрактный суперкласс, который подойдет как для метода смещений, так и для метода поиска:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/Page.java

```
public abstract class Page {

    public static enum SortDirection {
        ASC,
        DESC
    }

    protected int size = -1;                                  ← ❶ Вывод всех записей

    protected long totalRecords;                              ← ❷ Подсчет количества записей

    protected SingularAttribute sortAttribute;                ← ❸ Сортировка записей
    protected SortDirection sortDirection;

    protected SingularAttribute[] allowedAttributes;          ← ❹ Список допустимых
                                                                  для сортировки атрибутов
    // ...

    abstract public <T> TypedQuery<T> createQuery(
        EntityManager em,
        CriteriaQuery<T> criteriaQuery,
        Path attributePath
    );
}
```

❶ Модель хранит размер каждой страницы и количество записей на странице. Значение -1 означает, что будут возвращены все записи без ограничений.
❷ Хранение количества всех записей необходимо для некоторых вычислений, например чтобы понять, существует ли «следующая» страница.
❸ Постраничная выборка всегда требует строго определенного порядка следования записей. Как правило, сортировка осуществляется по конкретному атрибуту класса сущности в порядке возрастания или убывания. Поле `javax.persistence.metamodel.SingularAttribute` в JPA ссылается на атрибут сущности или встраиваемого класса; оно не может ссылаться на коллекцию (результаты запроса нельзя «упорядочить по коллекции»).
❹ Список `allowedAttributes` задается во время создания модели страницы. Он определяет допустимые для сортировки атрибуты, которые можно использовать в запросах.

Мы опустили некоторые тривиальные методы класса `Page` – в основном это методы чтения/записи. Однако подклассы должны реализовать абстрактный метод `createQuery()`: он описывает применение настроек страницы к запросу `CriteriaQuery` перед выполнением.

Сначала нужно связать интерфейс `Page` с уровнем хранения. Интерфейс DAO будет принимать экземпляр `Page`, когда потребуется осуществить постраничную выборку данных:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/ItemDAO.java

```
public interface ItemDAO extends GenericDAO<Item, Long> {

    List<ItemBidSummary> getItemBidSummaries(Page page);

    // ...

}
```

Таблица данных для отображения будет показывать список `List` оъектов передачи данных `ItemBidSummary`. Результат запроса не так важен в этом примере; мы могли также извлечь и список экземпляров `Item`. Ниже показана часть реализации DAO:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/ItemDAOImpl.java

```
public class ItemDAOImpl
    extends GenericDAOImpl<Item, Long>
    implements ItemDAO {

    // ...

    @Override
    public List<ItemBidSummary> getItemBidSummaries(Page page) {
        CriteriaBuilder cb =                    ← ❶ Запрос на основе критериев
            getEntityManager().getCriteriaBuilder();
        CriteriaQuery<ItemBidSummary> criteria =
            cb.createQuery(ItemBidSummary.class);

        Root<Item> i = criteria.from(Item.class);

        // Некоторые параметры запроса...
```

```
        // ...
        TypedQuery<ItemBidSummary> query =  ← ❷ Окончательная подготовка запроса
            page.createQuery(em, criteria, i);
        return query.getResultList();
    }
    // ...
}
```

❶ Это самый обычный запрос на основе критериев, который вы видели много раз до этого.
❷ Окончательная подготовка запроса ложится на плечи полученного объекта Page.

Конкретная реализация интерфейса Page подготавливает запрос, устанавливая необходимые смещение, количество извлекаемых записей и параметры поиска.

Реализация метода смещений

Ниже показана реализация стратегии постраничной выборки на основе смещений:

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/OffsetPage.java

```
public class OffsetPage extends Page {

    protected int current = 1;  ← ❶ Текущая страница

    // ...

    @Override
    public <T> TypedQuery<T> createQuery(EntityManager em,
                                         CriteriaQuery<T> criteriaQuery,
                                         Path attributePath) {
        throwIfNotApplicableFor(attributePath);  ← ❷ Попытка поиска атрибута для сортировки

        CriteriaBuilder cb = em.getCriteriaBuilder();

        Path sortPath = attributePath.get(getSortAttribute());  ← ❸ Добавит предложение
        criteriaQuery.orderBy(                                    ORDER BY
            isSortedAscending() ? cb.asc(sortPath) : cb.desc(sortPath)
        );

        TypedQuery<T> query = em.createQuery(criteriaQuery);

        query.setFirstResult(getRangeStartInteger());  ← ❹ Установка смещения

        if (getSize() != -1)  ← ❺ Установка количества результатов
            query.setMaxResults(getSize());

        return query;
    }
}
```

❶ Для постраничной выборки на основе смещений требуется знать номер текущей страницы. По умолчанию текущей является страница 1.

❷ Проверка возможности получения пути к атрибуту сортировки для данной страницы и, следовательно, к используемой запросом модели. Этот метод возбудит исключение, если атрибут сортировки недоступен в модели класса, на который ссылается запрос. Это – механизм повышения надежности, который выдаст осмысленное сообщение об ошибке, если попытаться связать неправильные настройки страницы с неправильным запросом.
❸ Добавление в запрос предложения `ORDER BY`.
❹ Установка смещения в запросе: первой выбираемой записи.
❺ Установка количества записей, выбираемых для данной страницы.

Мы показали реализацию не всех используемых методов. Например, здесь отсутствует такой метод, как `getRangeStartInteger()`, который вычисляет номер записи, первой в данной странице, в зависимости от размера страницы. Этот и другие вспомогательные методы вы найдете в исходном коде.

Обратите внимание, что порядок результатов может быть не определен: сортировка выполняется по наименованиям товаров в алфавитном порядке, и несколько товаров имеют одинаковое наименование, база данных вернет их в том порядке, который создатели СУБД посчитали приемлемым. Вы должны сортировать либо по уникальному ключевому атрибуту, либо добавить дополнительный критерий упорядочения по ключевому атрибуту. Несмотря на то что большинство разработчиков просто игнорирует проблему неопределенности при сортировке в методе смещений, предсказуемость порядка сортировки в методе поиска является обязательной.

Реализация метода поиска

Для реализации постраничной выборки на основе поиска нужно добавить в запрос ограничения. Предположим, что предыдущая страница показывала товары, отсортированные по наименованиям в алфавитном порядке, до значения «Coffee Machine», как показано на рис. 19.2, и нужно отобразить следующую страницу с помощью запроса SQL. Запомнив последнее значение на предыдущей странице – запись с «Coffe Machine» – и идентификатор (допустим 5), можно написать следующий код SQL:

```
select i.* from ITEM i
where
    i.NAME >= 'Coffee Machine'
    and (
        i.NAME <> 'Coffee Machine'
        or i.ID > 5
    )
order by
    i.NAME asc, i.ID asc
```

Первое ограничение гласит: «Верни все товары, наименование которых больше либо равно [Coffee Machine]», что приведет к поиску вперед до конца предыдущей страницы. База данных может эффективно реализовать данное ограничение с помощью поиска по индексу. Затем накладывается дополнительное ограничение, ис-

ключающее товар «Coffee Machine», пропуская таким образом запись, показанную на предыдущей странице.

Но в базе данных могут оказаться два товара с наименованием «Coffee Machine». Чтобы данные не потерялись при переходе между страницами, нужно использовать уникальный ключ. Вы должны упорядочить и ограничить результаты, используя этот уникальный ключ. Здесь используется первичный ключ, который гарантирует, что база данных вернет товары, наименование которых *не* «Coffee Machine», *или* товары (пусть даже с наименованием «Coffee Machine»), значение идентификатора которых больше показанного на предыдущей странице.

Конечно, если наименование товара (или значение другого столбца, по которому производится сортировка) уникально, дополнительный уникальный ключ можно не использовать. Код примера с обобщенным решением предполагает, что всегда будет применяться явный уникальный ключ. Также обратите внимание: тот факт, что идентификаторы товаров представляют собой возрастающую числовую последовательность, не играет никакой роли; самое главное, чтобы ключ гарантировал предсказуемый порядок сортировки.

Запрос можно переписать в более компактной форме, используя синтаксис конструктора строк значений:

```
select i.* from ITEM i
where
     (i.NAME, i.ID) > ('Coffee Machine', 5)
order by
    i.NAME asc, i.ID asc
```

Такое ограничение работает даже с JPQL в Hibernate. Однако в JPA эта возможность не стандартизована; ее нельзя использовать в запросах на основе критериев, и она поддерживается не всеми базами данных. Мы предпочитаем более длинный вариант, работающий везде. Чтобы выполнить сортировку по убыванию, оператор *больше, чем* следует заменить на *меньше, чем*.

Следующий код реализации класса `SeekPage` добавляет данное ограничение в запрос на основе критериев.

Файл: /apps/app-web/src/main/java/org/jpwh/web/dao/SeekPage.java

```
public class SeekPage extends Page {

    protected SingularAttribute uniqueAttribute;   ← ❶ Добавление уникального атрибута сортировки

    protected Comparable lastValue;   ← ❷ Запоминает значения на предыдущей странице
    protected Comparable lastUniqueValue;

    // ...

    @Override
    public <T> TypedQuery<T> createQuery(EntityManager em,
                                         CriteriaQuery<T> criteriaQuery,
                                         Path attributePath) {
```

```
        throwIfNotApplicableFor(attributePath);

        CriteriaBuilder cb = em.getCriteriaBuilder();

        Path sortPath = attributePath.get(getSortAttribute());  ← ❸ Сортировка результатов
        Path uniqueSortPath = attributePath.get(getUniqueAttribute());
        if (isSortedAscending()) {
            criteriaQuery.orderBy(cb.asc(sortPath), cb.asc(uniqueSortPath));
        } else {
                criteriaQuery.orderBy(cb.desc(sortPath),
                    cb.desc(uniqueSortPath));
            }

        applySeekRestriction(em, criteriaQuery, attributePath);  ← ❹ Добавление
                                                                    ограничений
        TypedQuery<T> query = em.createQuery(criteriaQuery);

        if (getSize() != -1)  ← ❺ Установка количества результатов
            query.setMaxResults(getSize());
        return query;
    }

    // ...
}
```

❶ В дополнение к обычному атрибуту сортировки методу поиска нужно передать атрибут, представляющий уникальный ключ. Это может быть любой уникальный атрибут модели сущности, но обычно выбирается атрибут первичного ключа.

❷ Для обоих атрибутов – сортировки и уникального ключа – нужно запомнить их значения на «предыдущей» странице. После этого можно будет извлечь данные для следующей страницы, выполнив поиск этих значений. Подойдет любой тип, реализующий интерфейс `Comparable`, как того требуют запросы на основе критериев.

❸ Результаты всегда должны сортироваться по атрибуту сортировки и атрибуту уникального ключа.

❹ Нужно добавить дополнительные ограничения (не показанные здесь) в предложение `WHERE`, чтобы поиск начинался после последних сохраненных значений.

❺ Нужно отсечь лишние результаты, согласно размеру страницы.

Целиком метод `applySeekRestriction()` можно найти в коде примеров; это код запроса на основе критериев, для которого здесь не нашлось места. Итоговый запрос эквивалентен версии SQL, показанной ранее.

Давайте протестируем новую функциональность уровня хранения для постраничной выборки.

Поиск границ страницы в методе поиска

Ранее мы говорили, что будет непросто реализовать в методе поиска переход к конкретной странице, поскольку для конкретной страницы неизвестны последние значения для поиска. Но эти значения можно найти с помощью выражения SQL, как показано ниже:

```
select i.NAME, i.ID
from ITEM i
where
    (select count(i2.*)
      from ITEM i2
      where (i2.NAME, i2.ID) <= (i.NAME, i.ID)
    ) % :pageSizeParameter = 0
order by i.NAME asc, i.ID asc
```

Этот запрос применяет оператор деления по модулю (%) для получения всех пар (NAME, ID), находящихся на границах страниц: они являются последними значениями на каждой странице. Аналогичный запрос на основе критериев показан в коде примеров.

19.2.3. Постраничная выборка

Теперь, вызывая метод ItemDAO#getItemBidSummaries(), вы должны передать экземпляр Page. Служба или пользовательский интерфейс, находящиеся выше уровня хранения, выполнят следующий код:

Файл: /apps/app-web/src/test/java/org/jpwh/test/service/PagingTest.java

```
OffsetPage page = new OffsetPage(
    3,                    ← Размер страницы
    itemDAO.getCount(),   ← Количество всех записей
    Item_.name, ASC,      ← Атрибут сортировки и ее направление по умолчанию
    Item_.name, Item_.auctionEnd, Item_.maxBidAmount
);                                          Все доступные для сортировки атрибуты
                                                для данной страницы
List<ItemBidSummary> result = itemDAO.getItemBidSummaries(page);  ←
```

Перейти к следующей странице очень просто – нужно лишь указать ее номер:

```
page.setCurrent(2);
result = itemDAO.getItemBidSummaries(page);
```

Методу поиска потребуется больше информации. Сначала нужно создать экземпляр SeekPage с дополнительным уникальным атрибутом:

Файл: /apps/app-web/src/test/java/org/jpwh/test/service/PagingTest.java

```
SeekPage page = new SeekPage(
    3,
    itemDAO.getCount(),
    Item_.name, ASC,
    Item_.id,   ← Дополнительный уникальный атрибут для поиска и упорядочения
    Item_.name, Item_.auctionEnd, Item_.maxBidAmount
);

List<ItemBidSummary> result = itemDAO.getItemBidSummaries(page);
```

Чтобы перейти к конкретной странице в методе смещений, нужно знать лишь номер страницы. Для стратегии поиска необходимо запоминать показанные на предыдущей странице значения:

```
ItemBidSummary lastShownOnPreviousPage = // ...

page.setLastValue(lastShownOnPreviousPage.getName());
page.setLastUniqueValue(lastShownOnPreviousPage.getItemId());

result = itemDAO.getItemBidSummaries(page);
```

Очевидно, это требует от клиента уровня хранения большего количества работы. Вместо простого числа он должен запоминать последние значения атрибута сортировки и уникального ключа.

Пользовательские интерфейсы, которые вы видели на снимках экрана, выполнены с помощью JSF и CDI и содержат необходимый код для взаимодействия с реализацией постраничной выборки. Пусть код примеров вдохновит вас на интеграцию постраничной выборки с вашими службами и пользовательскими интерфейсами.

Даже если не использовать JSF, все равно можно очень просто адаптировать решение для сортировки и постраничной выборки к другим фреймворкам веб-приложений. Если же вы намерены использовать JSF, следующий раздел как раз для вас: мы рассмотрим сложные варианты использования, такие как размещение ставки и редактирование информации о товаре, реализованные с помощью веб-интерфейса JSF поверх уровня службы с состоянием.

19.3. Создание JSF-приложений

В предыдущем примере можно было щелкнуть на названии товара в каталоге. Это действие открывает страницу с более детальной информацией о товаре, как показано на рис. 19.3.

Рис. 19.3 ❖ Просмотр данных о товаре и размещение ставки

В странице с описанием товара имеется форма, позволяющая сделать ставку. Это будет первый вариант использования, который мы реализуем: размещение ставки за товар. Для этого можно использовать простую службу с областью видимости запроса.

19.3.1. Службы с областью видимости запроса

Начнем с рассмотрения важного фрагмента шаблона XHTML для этой страницы:

Файл: /apps/app-web/src/main/webapp/auction.xhtml

```
<f:metadata>
    <f:viewParam name="id" value="#{auctionService.id}"/>
</f:metadata>
```

Эту страницу можно добавить в закладки, т. е. она может и будет вызываться с параметром запроса – идентификатором товара `Item`. Благодаря метаданным шаблон JSF свяжет значение параметра с полем `id` экземпляра `AuctionService` на стороне сервера. Мы реализуем эту службу чуть ниже.

На странице нужно отобразить некоторую информацию о товаре, запросив ее у службы `AuctionService`:

Файл: /apps/app-web/src/main/webapp/auction.xhtml

```
<h:outputText value="Auction End:"/>
<h:outputText value="#{auctionService.item.auctionEnd}">
    <f:convertDateTime pattern="dd. MMM yyyy HH:mm"/>
</h:outputText>

<h:outputText value="Starting Price:"/>
<h:outputText value="#{auctionService.item.initialPrice}"/>
```

Отображение этих данных потребует повторного вызова метода `AuctionService#getItem()`; не забудьте об этом во время реализации службы. Наконец, ниже показана форма размещения ставки:

Файл: /apps/app-web/src/main/webapp/auction.xhtml

```
<h:form>
    <h:inputHidden value="#{auctionService.id}"/>          ← ❶ Посылает идентификатор

    <h:inputText value="#{auctionService.newBidAmount}"
                 size="6"/>                                ← ❷ Задает значение ставки

    <h:commandButton value="Place Bid"
     action="#{auctionService.placeBid}"/>                 ← ❸Размещает ставку
</h:form>
```

❶ Вместе с формой нужно также передать идентификатор товара. Служба на стороне сервера действует в области видимости запроса, поэтому его экземпляр будет создаваться для каждого запроса отдельно: этот код вызовет метод `AuctionService#setId()`.

❷ Введенное значение ставки JSF установит с помощью метода `AuctionService#setNewBidAmount()`, когда будут обработаны параметры `POST`-запроса с формой.

❸ После связывания всех значений будет вызван метод `AuctionService#placeBid()`, указанный в атрибуте `action` формы.

Теперь рассмотрим класс службы, который является обычным компонентом, управляемым механизмом CDI:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/AuctionService.java

```
@Named
@RequestScoped   ← ❶ Не нужно хранить состояния
public class AuctionService {

    @Inject
    ItemDAO itemDAO;

    @Inject
    BidDAO bidDAO;
    long id;     ← ❷ Определение состояния
    Item item;
    BigDecimal highestBidAmount;
    BigDecimal newBidAmount;

    // ...
}
```

❶ В данном варианте использования не потребуется сохранять состояние между запросами. Экземпляр службы будет создан после отображения страницы аукциона `GET`-запросом, а JSF свяжет параметр запроса, вызвав метод `setId()`. Экземпляр службы будет уничтожен по окончании отображения. В промежутке между запросами сервер не будет хранить никаких данных. После отправки формы и начала обработки `POST`-запроса JSF вызовет `setId()`, чтобы связать скрытое поле формы, и вы сможете заново инициализировать состояние службы.

❷ Состояние, сохраняемое между запросами, – это значение идентификатора экземпляра `Item`, с которым работает пользователь, сам экземпляр `Item` после загрузки, текущая максимальная ставка за этот товар и новая ставка, введенная пользователем.

Не забывайте, что в JSF любой метод доступа к свойству, указанный в шаблоне XHTML, может вызываться несколько раз. В службе `AuctionService` данные будут загружаться при вызове методов доступа, поэтому вы должны предотвратить многократное обращение к базе данных по неосторожности:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/AuctionService.java

```
public class AuctionService {

    public void setId(long id) {
        this.id = id;
        if (item == null) {
            item = itemDAO.findById(id);
            if (item == null)
                throw new EntityNotFoundException();
```

```
            highestBidAmount = itemDAO.getMaxBidAmount(item);
        }
    }

    // Прочие методы чтения/записи...

}
```

Когда JSF вызовет `setId()` в первый раз, вы загрузите экземпляр `Item` с данным идентификатором только один раз. Если экземпляр сущности не найден, завершайте работу с ошибкой. Для полной инициализации состояния службы перед отображением данных дополнительно загружается максимальная ставка или вызывается метод действия.

Обратите внимание, что метод выполняется вне транзакции! В отличие от методов EJB, простому компоненту CDI транзакция не требуется. Объект `EntityManager` и контекст хранения, используемый в DAO, рассинхронизированы, и данные читаются из базы в режиме автоматического подтверждения (auto-commit). Даже если позднее, продолжая обрабатывать тот же запрос, вызвать транзакционный метод, будет использован тот же самый рассинхронизированный `EntityManager` в области видимости запроса.

Таким транзакционным методом является `placeBid()`:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/AuctionService.java

```
public class AuctionService {

    // ...

    @Transactional  ← ❶ Поместит этот метод в контекст транзакции
    public String placeBid() {
        itemDAO.joinTransaction();  ← ❷ Сохранение ставки

        if (!getItem().isValidBidAmount(  ← ❸ Сравнение с максимальной ставкой
                getHighestBidAmount(),
                getNewBidAmount()
            )) {
            ValidationMessages.addFacesMessage("Auction.bid.TooLow");
            return null;
        }

        itemDAO.checkVersion(getItem(), true);  ← ❹ Принудительное увеличение номера версии

        bidDAO.makePersistent(new Bid(getNewBidAmount(), getItem()));

        return "auction?id=" + getId() + "&faces-redirect=true";  ← ❺ Перенаправление
    }

}
```

❶ Аннотация `@Transactional` появилась в Java EE 7 (в версии JTA 1.2) и выполняет ту же функцию, что и аннотация `@TransactionAttribute` в компонентах EJB. Определяемый ею перехватчик поместит метод в контекст системной транзакции, по аналогии с методами EJB.

❷ Выполнение действий в рамках транзакции и сохранение новой ставки в базу данных для предотвращения состояния гонки. Контекст хранения следует присоединить к транзакции. Не имеет значения, какой экземпляр DAO вызывается: они все используют общий экземпляр `EntityManager` в области видимости запроса.
❸ Если, пока пользователь раздумывает, глядя на страницу, будет подтверждена другая транзакция, с более высокой ставкой, выполнение завершится с ошибкой, и страница аукциона будет показана заново с сообщением об ошибке.
❹ Для предотвращения состояния гонки необходимо также принудительно увеличить номер версии экземпляра `Item` перед выталкиванием контекста хранения. Если параллельно будет выполняться другая транзакция, которая загрузит из базы тот же экземпляр `Item` с такой же версией и текущей максимальной ставкой в методе `setId()`, одна из них завершится с ошибкой в методе `placeBid()`.
❺ Это простое перенаправление после обработки `POST`-запроса в JSF, чтобы пользователи могли безопасно загрузить страницу после сохранения ставки.

После выхода из метода `placeBid()` транзакция подтверждается, а присоединенный контекст хранения автоматически выталкивает изменения в базу данных. В Java EE 7, наконец, стала доступна удобная возможность EJB: декларативное описание границ транзакций, независимое от EJB.

Другой особенностью Java EE является область видимости диалога в CDI с возможностью интеграции с JSF.

19.3.2. Службы с областью видимости диалога

В Java EE можно отмечать компоненты и создаваемые экземпляры аннотацией `@ConversationScoped`, задавая область видимости диалога. Эта особая область видимости управляется вручную с помощью стандартного интерфейса `javax.enterprise.context.Conversation`. Чтобы понять, как работает область видимости диалога, мы советуем поразмышлять над примерами из предыдущей главы.

Самым коротким диалоговым взаимодействием (единицей работы с точки зрения пользователя приложения) является простой цикл запрос/ответ. Пользователь отправляет запрос, а сервер создает контекст диалога для хранения данных этого запроса. После отправки ответа это кратковременное диалоговое взаимодействие завершится и будет закрыто. То есть область видимости временного диалогового взаимодействия тождественна области видимости запроса; оба контекста имеют одинаковый жизненный цикл. Именно так спецификация CDI связывает область видимости диалога с запросом сервлета и, следовательно, с запросами в JSF и JAX-RS.

С помощью интерфейса `Conversation` можно продлить время жизни диалогового взаимодействия на сервере, и оно уже не будет временным. Если диалоговое взаимодействие продлить во время обработки запроса, во время следующего запроса можно будет задействовать тот же контекст диалогового взаимодействия и использовать его для передачи данных между запросами. Обычно для этого применяется контекст сеанса, но в этом случае приходится вручную разделять параллельные диалоговые взаимодействия (например, когда пользователь открыл две вкладки браузера). Как раз для преодоления этих сложностей был создан контекст

диалога, автоматически изолирующий данные в управляемом контексте, который гораздо удобнее обычного контекста сеанса.

Контекст диалога реализован на стороне сервера с помощью сеанса и хранит свои данные в пользовательском сеансе. Для изоляции и идентификации диалоговых взаимодействий внутри сеанса каждому из них присваивается идентификатор – его значение нужно отправлять с каждым запросом. Для этих целей даже был стандартизован параметр `cid`. Каждое диалоговое взаимодействие имеет также собственное время ожидания, чтобы сервер мог освободить ресурсы, если пользователь перестанет присылать запросы. Это позволяет серверу автоматически очищать устаревшие диалоги, не дожидаясь, пока истечет время жизни сеанса.

ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ:
А что насчет области видимости потока в JSF?

Область видимости потока (flow scope) была добавлена в последних версиях JSF и напоминает область видимости диалога CDI. Она отвечает за группировку и обнаружение шаблонов представления в рамках последовательности действий, а также за автоматизацию правил навигации в этой последовательности. Ее также можно использовать для передачи состояния между запросами. К сожалению, время ожидания в рамках сеанса для контекста потока не определено; также отсутствует API для удаления или продления жизни контекста. Мы считаем контекст потока непригодным к использованию в его текущей реализации, за исключением простейших вариантов использования, в которых избыточное потребление ресурсов на сервере не играет роли. Мы надеемся, что в будущих версиях Java EE функциональность диалоговых взаимодействий CDI и контекстов потока JSF будет согласована и объединит сильные стороны обеих.

Другой приятной особенностью контекста длительных (не временных) диалоговых взаимодействий является автоматическая защита от параллельного доступа на стороне сервера: если пользователь быстро нажмет кнопку несколько раз подряд, где каждый запрос будет содержать значение идентификатора конкретного диалогового взаимодействия, сервер прекратит обработку всех запросов, кроме одного, возбудив исключение `BusyConversationException`.

Сейчас мы рассмотрим пример использования области видимости диалога в JSF. Мы реализуем вариант использования с последовательностью действий, требующей нескольких запросов для завершения: выставление товара на аукцион.

Последовательность действий «редактирование информации о товаре»

Немного поразмыслив, можно заметить, что создание и редактирование информации о товаре – очень похожие задачи. Эти последовательности действий являются диалоговыми взаимодействиями – единицами работы с точки зрения пользователя приложения, поэтому пользователь ожидает, что и выглядеть они будут похоже. Соответственно, мы постараемся не допустить дублирования кода;

реализуем оба варианта использования с помощью одного пользовательского интерфейса и одной службы на стороне сервера. Приложение должно направлять пользователя через последовательность действий; на рис. 19.4 показана диаграмма состояний этой последовательности.

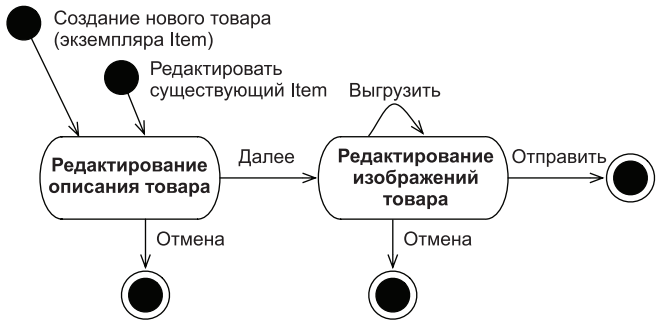


Рис. 19.4 ❖ Последовательность действий в диалоговом взаимодействии по редактированию товара

Расшифруем ее. Пользователь может начать диалоговое взаимодействие без всяких данных, если товар отсутствует. С другой стороны, пользователь может иметь идентификатор существующего товара, который, скорее всего, будет обычным числом. Не важно, как пользователь узнал этот идентификатор; возможно, он выполнил поиск в предыдущем диалоговом взаимодействии. Это распространенный сценарий, и, как многие последовательности диалоговых взаимодействий, он имеет несколько точек входа (закрашенные кружки на диаграмме состояний).

С точки зрения пользователя редактирование товара представляет собой многошаговый процесс: каждый прямоугольник с закругленными краями представляет состояние приложения. В состоянии редактирования товара приложение показывает пользователю диалог или форму, где пользователь может менять описание товара или начальную цену. Тем временем приложение будет ждать событий от пользователя; мы называем это *временем раздумий пользователя*.

После того как от пользователя поступит событие «Далее», приложение обработает его, а пользователь перейдет к следующему состоянию (ожидания) – редактированию изображений. По завершении редактирования изображений товара пользователь завершает диалоговое взаимодействие, отправляя товар и его изображения на сервер (обведенный закрашенный круг). Нажатие кнопки **Cancel** (Отмена) на любой странице завершает взаимодействие. С точки зрения пользователя приложения все диалоговое взаимодействие целиком является атомарной единицей: после нажатия кнопки **Submit** (Отправить) все изменения будут сохранены. Никакой другой переход между состояниями не должен сохранять изменений в базу данных.

Мы реализуем эту последовательность в стиле мастера редактирования, состоящего из двух веб-страниц. На первой странице (рис. 19.5) пользователь будет вводить или редактировать описание товара.

Рис. 19.5 ❖ Первая страница мастера редактирования информации о товаре

Обратите внимание, что отображение этой страницы не создает контекста длительного диалогового взаимодействия на сервере. Это была бы напрасная трата ресурсов, поскольку еще неизвестно, захочет ли пользователь изменить информацию о товаре. Создание контекста длительного диалогового взаимодействия и передача данных между запросами произойдут после первого нажатия кнопки **Next** (Далее). Если проверка формы пройдет успешно, сервер сохранит данные диалогового взаимодействия в сеансе пользователя и направит его на страницу редактирования изображения (рис. 19.6).

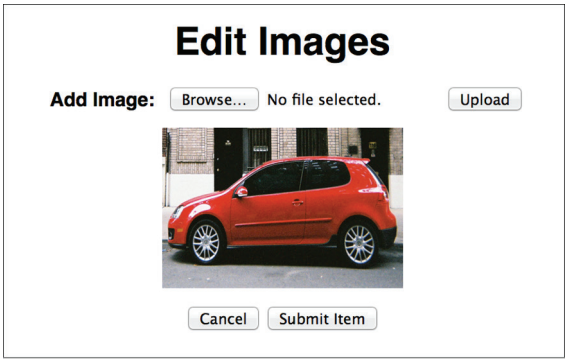


Рис. 19.6 ❖ Вторая страница мастера для редактирования изображений товара

Единица работы закончится, когда пользователь отправит данные о товаре после выгрузки изображений. Пользователь может в любой момент нажать на кнопку **Cancel** (Отмена) и завершить взаимодействие или отойти от терминала, из-за чего диалоговое взаимодействие завершится после определенного времени ожидания. Пользователь может работать в нескольких вкладках браузера параллельно, запуская сразу несколько диалоговых взаимодействий; следовательно, в сеансе пользователя на сервере они должны быть изолированы. Все данные диалогового взаимодействия будут удалены, если время жизни сеанса пользователя превысит время ожидания (или сервер кластера с привязанным сеансом отключится).

В шаблонах XHTML для этого мастера нет никакой специальной разметки, относящейся к диалоговому взаимодействию; оно автоматизируется с помощью JSF и CDI. Если во время отображения формы обнаружится контекст длительного

диалогового взаимодействия, его идентификатор будет автоматически отправлен вместе с данными формы JSF в виде значения скрытого поля. В JSF это работает даже для перенаправлений; параметр `cid` будет автоматически добавляться к URL назначения перенаправления.

Служба с областью видимости диалога связана с этими страницами на стороне сервера; все действия происходят в службе `EditItemService`, как и управление контекстом.

Контекст хранения в диалоговом взаимодействии

Служба `EditItemService` обеспечивает работу мастера и весь рабочий процесс на стороне сервера:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
@Named
@ConversationScoped   <-- ❶ Поведение соответствует области видимости запроса
public class EditItemService implements Serializable {   <-- ❷ Должен быть сериализуемым

    @Inject           <-- ❸ Может быть сериализован
    ItemDAO itemDAO;

    @Inject
    ImageDAO imageDAO;

    // ...

    @Inject           <-- ❹ Вызов Conversation API
    Conversation conversation;

    Long itemId;      <-- ❺ Состояние службы
    Item item = new Item();

    transient Part imageUploadPart;          <-- ❻ Временное состояние службы

    public void setItemId(Long itemId) {     <-- ❼ Загрузка товара
        this.itemId = itemId;
        if (item.getId() == null && itemId != null) {
            item = itemDAO.findById(itemId);  <-- ❽ Извлечение товаров из базы данных
            if (item == null)
                throw new EntityNotFoundException();
        }
    }

    // ...

}
```

❶ Экземпляр службы находится в области видимости диалога. Диалоговый контекст является по умолчанию временным, поэтому служба ведет себя, как если бы находилась в области видимости запроса.

❷ В отличие от версии с областью видимости запроса, здесь служба должна реализовать интерфейс `Serializable`. Экземпляр `EditItemService` может быть сохранен в сеансе HTTP, а этот сеанс может быть сериализован и записан на диск или отправлен по сети кластера.

В главе 18 мы пошли по легкому пути, используя компонент EJB с состоянием, сказав: «Пассивирование для данного компонента недоступно». Все, что находится в области видимости диалогового взаимодействия CDI, должно иметь возможность пассивирования, а следовательно, поддерживать сериализацию.

❸ Внедренные экземпляры DAO обладают зависимой областью видимости и также являются сериализуемыми. Может показаться, что это не так, поскольку каждый из них имеет поле типа `EntityManager`, который нельзя сериализовать. Через мгновение мы разберемся с этим нюансом.

❹ Реализация `Conversation` API будет предоставлена контейнером. Он необходим для управления контекстом диалога и понадобится нам, когда пользователь первый раз нажмет кнопку **Next** (Далее) и временный контекст нужно будет сделать продолжительным.

❺ Это состояние службы: товар, редактируемый пользователем на страницах мастера. Работа начинается с нового экземпляра сущности `Item` во временном состоянии. Если эта служба инициализируется значением идентификатора, экземпляр `Item` будет загружен в методе `setItemId()`.

❻ А это временное состояние службы. Оно нужно лишь ненадолго, когда пользователь нажмет кнопку **Upload** (Выгрузить) на странице редактирования изображений товара. Класс `Part` из спецификации `Servlet` API не сериализуется. В службах области видимости диалога часто можно увидеть временное состояние, но его следует инициализировать в каждом запросе, если оно необходимо.

❼ Метод `setItemId()` вызывается, только если в запросе передан идентификатор товара. Следовательно, это диалоговое взаимодействие имеет две точки входа: с использованием идентификатора товара и без.

❽ Если пользователь редактирует информацию о товаре, нужно загрузить ее из базы данных. Нам по-прежнему приходится полагаться на контекст хранения из области видимости запроса, поэтому, как только обработка запроса закончится, экземпляр `Item` окажется в отсоединенном состоянии. Можно хранить отсоединенные экземпляры сущностей в виде состояния службы области видимости диалога, выполняя слияние, когда потребуется сохранить изменения в базу данных (см. раздел 10.3.4).

В отличие от компонентов EJB с состоянием, в данном случае нельзя отключить пассивирование контекста диалога. Спецификация CDI требует, чтобы класс и все зависимости компонента области видимости диалога, не являющиеся временными, поддерживали сериализацию. На самом деле вы получите ошибку развертывания, если нечаянно поместите в область видимости диалога объект, который нельзя сериализовать.

Для обхода этой проверки мы сказали выше, что `GenericDAO` реализует `java.io.Serializable`. Но поле `EntityManager` класса `GenericDAOImpl` *не* сериализуется! Однако этот код работает, потому что CDI использует *контекстные ссылки* – «умные» объекты-заменители.

Во время выполнения поле типа `EntityManager` в классе DAO не является подлинным экземпляром контекста хранения. Оно ссылается на некоторый объект `EntityManager` – некоторый текущий контекст хранения. Не забывайте, что CDI создаст и внедрит зависимость через конструктор. Но, поскольку она объявлена в области видимости запроса, во время выполнения вместо нее будет внедрен специальный прокси-объект, который выглядит как настоящий `EntityManager`. Прокси-объект

будет перенаправлять все вызовы настоящему объекту `EntityManager`, который он найдет в контексте текущего запроса. Прокси-объект *является* сериализуемым и не сохраняет ссылку на экземпляр `EntityManager` после окончания обработки запроса. Затем он может быть запросто сериализован; во время десериализации (возможно, в другой виртуальной машине JVM) он продолжит выполнять свою работу и, когда потребуется, найдет экземпляр `EntityManager` в контексте запроса. Следовательно, контекст хранения не будет сериализован целиком, а только прокси-объект, который сможет найти контекст хранения в текущем контексте запроса.

На первый взгляд, это может показаться странным, но так работает CDI: если компонент в области видимости запроса будет внедрен в контекст диалога, сеанса или приложения, он использует косвенную ссылку. Если попробовать обратиться к прокси-объекту `EntityManager` через DAO, в отсутствие активного контекста запроса (например, в методе сервлета `init()`), возникнет исключение `ContextNotActiveException`, поскольку прокси-объект не сможет получить текущего экземпляра `EntityManager`. Спецификация CDI также говорит, что такие прокси-объекты могут подвергаться пассивированию (сериализации), даже если представляемый ими компонент не обладает подобной возможностью.

Предположим, что пользователь заполнил форму с данными о товаре на первой странице мастера, а затем нажал кнопку **Next** (Далее). В этот момент важно не потерять временного контекста диалога, превратив его в длительный контекст.

Создание длительных диалоговых взаимодействий

Нажатие кнопки **Next** (Далее) на странице мастера отправит форму с данными объекта `Item` на сервер, и пользователь увидит страницу редактирования изображений. Поскольку служба `EditItemService` находится во временном контексте диалогового взаимодействия, обработка запроса начнется в новом временном контексте. Помните, что временный контекст диалогового взаимодействия – это аналог контекста запроса.

При обработке запроса необходимо настроить текущий временный контекст диалога с помощью `Conversation` API:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
public class EditItemService implements Serializable {

    // ...

    public String editImages() {
        if (conversation.isTransient()) {
            conversation.setTimeout(10 * 60 * 1000);  ← 10 минут
            conversation.begin();
        }
        return "editItemImages";
    }

    // ...

}
```

Метод действия будет вызван после сохранения экземпляра `Item` в поле `EditItemService#item` механизмом JSF. Чтобы сделать временное диалоговое взаимодействие длительным, нужно установить для него время ожидания. Очевидно, это время меньше либо равно времени жизни сеанса пользователя; большие значения не имеют смысла. Сервер сохранит состояние службы в сеансе пользователя и автоматически отобразит сгенерированный идентификатор диалогового взаимодействия в виде скрытого поля на странице редактирования изображений. Если потребуется, значение идентификатора взаимодействия можно получить с помощью метода `Conversation#getId()`. Вы даже можете установить собственное значение идентификатора с помощью метода `Conversation#begin()`.

Теперь сервер будет ждать следующего запроса с идентификатором диалогового взаимодействия, возможно отправленного со страницы редактирования изображений. Если ожидание продлится слишком долго из-за того, что у длительного диалогового взаимодействия или сеанса истекло время жизни, во время запроса будет возбуждено исключение `NonexistentConversationException`.

Если пользователь захочет загрузить изображение товара, следующий запрос вызовет соответствующий метод службы:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
public class EditItemService implements Serializable {

    // ...

    public void uploadImage() throws Exception {
        if (imageUploadPart == null)
            return;

        Image image =                          ← ❶ Создание экземпляра
            imageDAO.hydrateImage(imageUploadPart.getInputStream());
        image.setName(imageUploadPart.getSubmittedFileName());
        image.setContentType(imageUploadPart.getContentType());

        image.setItem(item);  ← ❷ Добавление временного объекта Image
        item.getImages().add(image);
    }

    // ...

}
```

❶ Создание экземпляра сущности `Image` из составных данных формы.

❷ Нужно добавить временный экземпляр `Image` в коллекцию изображений временного или отсоединенного экземпляра товара `Item`. Это диалоговое взаимодействие будет потреблять все больше и больше памяти на сервере, по мере добавления выгружаемых изображений в состояние диалогового взаимодействия и, следовательно, в сеанс пользователя.

Наиболее важными аспектами построения серверной системы с сохранением состояния являются количество потребляемой памяти и максимальное количество одновременно обрабатываемых пользовательских сеансов. С данными диало-

гового взаимодействия нужно обращаться осторожно. Всегда следует подумать, стоит ли сохранять данные, полученные от пользователя или из базы данных, в течение всего времени жизни диалогового взаимодействия.

Когда пользователь нажимает кнопку **Submit** (Отправить), диалоговое взаимодействие завершается, а все временные и отсоединенные экземпляры сущностей должны быть сохранены.

Завершение длительных диалоговых взаимодействий

Последовательность действий в диалоговом взаимодействии закончится, когда будет сохранен временный или отсоединенный экземпляр `Item` со всеми его изображениями:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
public class EditItemService implements Serializable {

    // ...

    @Transactional                                  ← ❶ Обертывает вызов метода
    public String submitItem() {
        itemDAO.joinTransaction();                  ← ❷ Присоединяет контекст хранения к транзакции

        // ...

        item = itemDAO.makePersistent(item);        ← ❸ Делает экземпляр Item хранимым
        if (!conversation.isTransient())            ← ❹ Завершение диалогового взаимодействия
            conversation.end();

        return "auction?id=" + item.getId() + "&faces-redirect=true";
    }                                                   ❺ Перенаправление

    // ...

}
```

❶ Вызов метода будет обернут системным перехватчиком транзакций.

❷ Чтобы сохранить данные, нужно присоединить рассинхронизированный контекст хранения из области видимости запроса к системной транзакции.

❸ Этот вызов DAO сохранит временный или отсоединенный экземпляр `Item`. Поскольку в аннотации `@OneToMany` описано правило каскадной передачи состояния, также будет сохранена коллекция временных или отсоединенных элементов `Item#images`. В соответствии с контрактом DAO в качестве текущего состояния следует использовать возвращаемый экземпляр.

❹ Завершение продолжительного диалогового взаимодействия вручную. Фактически это понижение статуса: продолжительное взаимодействие становится временным. После окончания обработки запроса контекст взаимодействия будет разрушен вместе с экземпляром службы. Состояние диалогового взаимодействия будет удалено из сеанса пользователя.

❺ Пользователь будет перенаправлен на страницу с описанием товара после обработки `POST`-запроса с помощью JSF; также будет отправлен идентификатор сохраненного экземпляра `Item`.

Кроме того, пользователь может в любой момент нажать кнопку **Cancel** (Отмена), чтобы закрыть мастера:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jsf/EditItemService.java

```
public class EditItemService implements Serializable {

    // ...

    public String cancel() {
        if (!conversation.isTransient())
            conversation.end();
        return "catalog?faces-redirect=true";
    }

    // ...
}
```

На этом мы заканчиваем пример реализации службы с состоянием, использующей JSF, CDI и уровень хранения на основе JPA. По нашему мнению, JSF и CDI отлично сочетаются с JPA; вы получаете хорошо протестированную и стандартизованную модель программирования, требующую небольших затрат ресурсов и количества строк кода.

Далее мы продолжим использовать CDI, но вместо JSF будем работать с архитектурой службы без состояния на основе JAX-RS и JPA, подходящей для любого толстого клиента. Одной из трудностей этой архитектуры является сериализация данных.

19.4. Сериализация данных предметной модели

Когда в разделе 3.2.3 мы впервые обсуждали создание классов с возможностью длительного хранения, то вкратце упомянули, что они не обязаны реализовать интерфейс `java.io.Serializable`. Этот интерфейс-маркер можно использовать, только когда он действительно необходим.

Один из таких вариантов был показан в главе 18. Экземпляры предметной модели передавались между EJB-системами клиента и сервера, подвергаясь автоматической сериализации в некоторый формат передачи данных на одном конце и десериализации на другом. Все это работало без сучка и задоринки, поскольку и клиент, и сервер были виртуальными машинами Java, библиотеки Hibernate были доступны в обеих системах. Клиент использовал удаленный вызов процедур (Remote Method Invocation, RMI) и стандартный формат сериализации Java (поток байтов, представляющий объекты Java).

Если ваш клиент не использует виртуальную машину Java, вам вряд ли захочется получать поток байтов, представляющий объекты Java на сервере. Распространен сценарий, где сервер без состояния обеспечивает работу толстого клиента, такого как приложение JavaScript, работающее в веб-браузере, или мобильное приложение. Обычно для такого сценария нужно реализовать на сервере интерфейс *Web API*, использующий HTTP для передачи данных в формате XML или JSON,

которые клиент должен будет анализировать. Клиент также будет посылать данные в формате XML или JSON, когда нужно будет сохранить данные на сервере, поэтому сервер должен уметь производить и потреблять данные в этих форматах.

Проектирование REST-приложений на основе гипермедиа

Приложение, обменивающееся данными в формате XML или JSON, многие называют системой в стиле *REST*. Но наиболее важным аспектом архитектуры с передачей состояния представления (REST) является обмен гипермедиадокументами между клиентом и сервером. Гипермедиадокументы содержат данные и возможные действия с этими данными: отсюда происходит название «Гипермедиа как механизм передачи состояния приложения» (Hypermedia As The Engine Of Application State, HATEOAS). Поскольку эта книга о Hibernate, а не о проектировании Web API, то мы покажем лишь, как создать простой интерфейс HTTP, обменивающийся обычными документами XML, который не использует гипермедиа и не спроектирован в стиле REST.

При проектировании своего API обратите внимание на *H-фактор* выбранного формата данных (см. http://amundsen.com/hypermedia/hfactor/) и прочтите замечательную книгу *RESTful Web APIs* (Richardson, 2013) Леонарда Ричардсона (Leonard Richardson), Майка Амундсена (Mike Amundsen) и Сэма Руби (Sam Ruby). Мы советуем избегать формата JSON, поскольку для улучшения своего H-фактора он требует использования нестандартных расширений.

Создание собственного гипермедиаформата на основе XML (возможно, путем расширения примера, показанного в этой главе) – тоже не лучшее решение. Мы предпочитаем обычный XHTML: у него отличный H-фактор, его легко писать и читать с помощью повсеместно доступных API. При сжатии он по своей эффективности может превзойти JSON; а во время разработки и тестирования с ним очень приятно работать интерактивно. Джон Мур (Jon Moore) предоставил прекрасный пример такого решения в статье «Building Hypermedia APIs with HTML» (www.infoq.com/presentations/web-api-html).

Сейчас мы напишем HTTP-сервер на основе фреймворка JAX-RS, который будет производить и потреблять документы XML. Хотя в примерах используется XML, они могут быть реализованы и с применением JSON, а фундаментальные проблемы, которые мы обсудим, будут в обоих случаях одинаковыми.

19.4.1. Создание JAX-RS-службы

Приступим к созданию службы JAX-RS. Один из ее методов будет отправлять документы XML, представляющие экземпляры сущностей `Item`, когда пользователь будет присылать `HTTP`-запрос GET. Другой метод будет принимать документ XML для обновления экземпляра Item запросом `PUT`:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jaxrs/ItemService.java

```
@Path("/item")                          ← ❶ Путь для запроса
public class ItemService {

    @Inject
    ItemDAO itemDAO;
```

```
    @GET                          ← ❷ Запрос GET
    @Path("{id}")                 ← ❸ Аргумент вызова
    @Produces(APPLICATION_XML)    ← ❹ Сериализует данные в XML
    public Item get(@PathParam("id") Long id) {
        Item item = itemDAO.findById(id);
        if (item == null)
            throw new WebApplicationException(NOT_FOUND);
        return item;
    }

    @PUT
    @Path("{id}")
    @Consumes(APPLICATION_XML)    ← ❺ Десериализация XML
    @Transactional                ← ❻ Начало транзакции
    public void put(@PathParam("id") Long id, Item item) {
        itemDAO.joinTransaction();
        itemDAO.makePersistent(item);
    }

    // ...
}
```

❶ Когда сервер получает запрос, путь в котором начинается с `/item`, он будет обработан методом этой службы. По умолчанию экземпляр службы находится в области видимости запроса, но вы можете использовать аннотации CDI, чтобы изменить это.
❷ HTTP-запрос `GET` обрабатывается этим методом.
❸ Контейнер передаст методу сегмент после `/item`, как, например, `/item/123`. Он отображается как параметр метода при помощи аннотации `@PathParam`.
❹ Этот метод возвращает данные в формате XML; следовательно, кто-то должен сериализовать возвращаемое значение в XML. Будьте бдительны: эта аннотация не то же самое, что аннотация продюсера в CDI. Она из другого пакета!
❺ Этот метод принимает данные в формате XML: следовательно, кто-то должен десериализовать XML-документ и преобразовать его в отсоединенный экземпляр `Item`.
❻ Поскольку этот метод сохраняет данные, нужно запустить системную транзакцию, присоединив к ней контекст хранения.

Стандарт JAX-RS описывает автоматическое преобразование (marshalling) для большинства важных видов информации и целого набора типов Java. К примеру, реализация JAX-RS должна уметь производить и потреблять XML-документы для классов, созданных с применением Java-архитектуры для связывания с XML (JAXB). Следовательно, класс сущности `Item` из модели предметной области должен стать классом JAXB.

19.4.2. Применение JAXB-отображений

JAXB использует аннотации для определения свойств класса, так же как JPA. Эти аннотации отображают поля класса в элементы и атрибуты документа XML. Среда выполнения JAXB автоматически преобразует сущности в формат XML и обратно. Вам это уже должно казаться знакомым; JAXB является отличным дополнением к предметной модели, использующей JPA.

Ниже показан простой документ XML, представляющий экземпляр `Item`:

```
<item id="1" auctionEnd="2018-03-06T15:00:00+01:00">
    <name>Baseball Glove</name>
    <description>It is brown.</description>
    <initialPrice>5.00</initialPrice>
    <bids>  ← А где продавец? Мы скоро это обсудим
        <bid id="1" createdOn="2018-03-06T15:01:00+01:00">
            <amount>11.00</amount>
        </bid>
        <bid id="2" createdOn="2018-03-06T15:02:00+01:00">
            <amount>12.00</amount>
        </bid>
        <bid id="3" createdOn="2018-03-06T15:03:00+01:00">
            <amount>13.00</amount>
        </bid>
    </bids>
</item>
```

Для получения такой схемы XML было принято несколько проектных решений. Давайте посмотрим на аннотации JAXB в классе `Item`, чтобы разобраться с имеющимися возможностями:

Файл: /apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```
@Entity
@XmlRootElement                          ← ❶ Отображение в XML
@XmlAccessorType(XmlAccessType.FIELD)    ← ❷ Использует поля класса
public class Item implements Serializable {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    @XmlAttribute
    protected Long id;

    @NotNull
    @Future(message = "{Item.auctionEnd.Future}")
    @XmlAttribute
    protected Date auctionEnd;

    // ...
}
```

❶ Экземпляр `Item` отображается в XML-элемент `<item>`. Фактически эта аннотация разрешает использование JAXB для этого класса.

❷ Во время сериализации и десериализации экземпляр JAXB должен обращаться напрямую к полям класса, а не к методам чтения/записи. Доводы в пользу такого решения те же, что и для JPA: это дает свободу при проектировании методов.

Идентификатор товара и дата окончания аукциона становятся атрибутами XML, а остальные поля отобразятся во вложенные XML-элементы. Не нужно от-

мечать аннотациями JAXB поля класса `description` и `initialPrice`, поскольку они будут отображены в элементы по умолчанию. Проще дело обстоит с атрибутами классов модели предметной области, хранящими единственное значение: они отображаются в XML-атрибуты или во вложенные XML-элементы. А что насчет коллекций и связей сущностей?

Можно просто добавить коллекцию со всеми элементами в документ XML, как показано на примере `Item#bids`:

Файл: /apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```
public class Item implements Serializable {

    @OneToMany(mappedBy = "item")
    @XmlElementWrapper(name = "bids")
    @XmlElement(name = "bid")
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Здесь есть возможность для оптимизации; если при возвращении товара `Item` потребуется включить все ставки, они все должны быть загружены. В настоящий момент в JPA понадобится выполнить несколько запросов для загрузки экземпляра `Item` и коллекции `Item#bids`, для которой настроена отложенная загрузка. При подготовке ответа сериализатор JAXB автоматически обойдет каждый элемент коллекции.

Hibernate инициализирует данные коллекции с помощью немедленной или отложенной загрузки, а JAXB (или любой другой сериализатор) сериализует каждый элемент по отдельности. Фактически Hibernate применяет специальные коллекции, как объяснялось в разделе 12.1.2, поэтому при сериализации никаких различий наблюдаться не будет. Это будет важно позднее, при десериализации XML, поэтому оставим пока этот вопрос.

Если не требуется отображать коллекцию или поле в документ XML, ее нужно отметить аннотацией `@XmlTransient`:

Файл: /apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```
public class Item implements Serializable {

    @OneToMany(mappedBy = "item", cascade = MERGE)
    @XmlTransient
    protected Set<Image> images = new HashSet<>();

    // ...
}
```

Работать с коллекциями довольно легко независимо от того, что они содержат: экземпляры простых типов, встраиваемых типов или множественные связи сущностей. Конечно, нужно быть осторожным при работе с циклическими ссылками (circular references), когда, например, объект `Bid` содержит обратную ссылку на

объект `Item`. В какой-то момент нужно остановиться и объявить ссылку несериализуемой (transient).

Самую большую трудность во время сериализации экземпляров сущностей, загруженных Hibernate, представляют внутренние прокси-объекты: заглушки для коллекций и связей с отложенной загрузкой. В классе `Item` таковым является поле `seller`, ссылающееся на сущность `User`.

19.4.3. Сериализация прокси-объектов Hibernate

Для поля `Item#seller` с помощью аннотации `@ManyToOne(fetch = LAZY)` настроена отложенная загрузка. В момент загрузки экземпляра сущности `Item` ее поле `seller` ссылается не на настоящий объект `User`: это прокси-класс `User`, который Hibernate сгенерировал во время выполнения.

Если не добавлять никаких настроек, JAXB отобразит это поле так:

```
<item id="1" auctionEnd="2018-03-06T15:00:00+01:00">
    <!-- ... -->
    <seller/>
    <!-- ... -->
</item>
```

Глядя на такой документ, клиент решит, что у товара нет продавца. Это, конечно же, неверно; неинициализированный прокси-объект – это *не* то же самое, что `null`! Вы можете придать пустому элементу XML особое значение, сообщая клиенту, что пустой элемент означает прокси-объект, а отсутствие элемента соответствует `null`. К сожалению, нам попадались такие решения сериализации (даже разработанные для работы с Hibernate), которые не делали этого различия. Некоторые решения, не предназначенные для Hibernate, наткнувшись на прокси-объект, могут даже аварийно завершить работу.

Как правило, вы должны настроить свой инструмент сериализации для осмысленной обработки прокси-объектов Hibernate. Для неинициализированных прокси-объектов в этом приложении требуется получить следующий фрагмент XML:

```
<item id="1" auctionEnd="2018-03-06T15:00:00+01:00">
    <!-- ... -->
    <seller type="org.jpwh.web.model.User" id="123"/>
    <!-- ... -->
</item>
```

Это та же информация, что содержится в прокси-объекте: класс сущности и идентификатор экземпляра, представляемого прокси-объектом. Теперь клиент знает, что у товара действительно есть продавец с определенным идентификатором; он сможет запросить эти данные, если потребуется. Получив этот документ XML на сервере, когда пользователь будет редактировать информацию о товаре, вы сможете восстановить прокси-объект, зная идентификатор и класс сущности.

Вы должны определить класс предметной модели, представляющий подобную ссылку на сущность, и отобразить его в элементы и атрибуты XML:

Файл: /apps/app-web/src/main/java/org/jpwh/web/model/EntityReference.java

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class EntityReference {

    @XmlAttribute
    public Class type;

    @XmlAttribute
    public Long id;

    public EntityReference() {
    }

    public EntityReference(Class type, Long id) {
        this.type = type;
        this.id = id;
    }
}
```

Далее вы должны настроить преобразование экземпляра `Item`, чтобы для поля `Item#seller` вместо реального объекта `User` использовался объект `EntityReference`. Для такого поля в JAXB нужно указать собственный адаптер:

Файл: /apps/app-web/src/main/java/org/jpwh/web/model/Item.java

```
public class Item implements Serializable {

    @NotNull
    @ManyToOne(fetch = LAZY)
    @XmlJavaTypeAdapter(EntityReferenceAdapter.class)
    protected User seller;

    // ...
}
```

Вы можете использовать `EntityReferenceAdapter` для любого поля, представляющего связь между сущностями. Он знает, как читать и записывать объекты `EntityReference` из XML и обратно:

Файл: /apps/app-web/src/main/java/org/jpwh/web/jaxrs/EntityReferenceAdapter.java

```
public class EntityReferenceAdapter
    extends XmlAdapter<EntityReference, Object> {
    EntityManager em;

    public EntityReferenceAdapter() {          ← ❶ Записывает экземпляры EntityReference
    }
    public EntityReferenceAdapter(EntityManager em) {   ← ❷ Читает экземпляры EntityReference
```

```
        this.em = em;
    }

    @Override
    public EntityReference marshal(Object entityInstance)
        throws Exception {

        Class type = getType(entityInstance);  ← ❸ Создает представление для сериализации
        Long id = getId(type, entityInstance);
        return new EntityReference(type, id);
    }

    @Override
    public Object unmarshal(EntityReference entityReference)
        throws Exception {
        if (em == null)
            throw new IllegalStateException(
                "Call Unmarshaller#setAdapter() and " +
                    "provide an EntityManager"
            );

        return em.getReference(                ← ❹ Создает прокси-объект
            entityReference.type,
            entityReference.id
        );
    }
}
```

❶ JAXB вызовет этот конструктор, когда понадобится сгенерировать документ XML. В этом случае экземпляр `EntityManager` не понадобится: прокси-объект уже содержит всю информацию, необходимую для создания экземпляра `EntityReference`.

❷ JAXB должен вызвать этот конструктор во время чтения документа XML. Вам понадобится экземпляр `EntityManager`, чтобы получить прокси-объект Hibernate из экземпляра `EntityReference`.

❸ При записи в документ XML возьмите прокси-объект Hibernate и создайте сериализуемое представление. Здесь будут вызваны внутренние методы Hibernate, которые в этом примере не показаны.

❹ При чтении документа XML возьмите сериализуемое представление и создайте прокси-объект Hibernate, связанный с текущим контекстом хранения.

Наконец, вам понадобится расширение для JAX-RS, которое автоматически инициализирует адаптер текущим экземпляром `EntityManager` из области видимости запроса, когда нужно будет преобразовать XML-документ на сервере. Вы можете найти расширение `EntityReferenceXMLReader` в коде примеров.

Осталось обсудить несколько моментов. Во-первых, мы еще не говорили о преобразовании коллекций. Во время вызова службы любой элемент `<bids>` в документе XML будет десериализован, и на основе этих данных созданы отсоединенные экземпляры `Bid`. Во время работы службы они будут доступны в отсоединенной коллекции `Item#bids`. Больше ничего не произойдет, да и не может произойти:

коллекция, созданная JAXB после преобразования документа, не будет одной из специальных коллекций Hibernate. Даже если в отображении вы настроили для коллекции `Item#bids` каскадное слияние, метод `EntityManager#merge()` проигнорирует его.

Это напоминает проблему с прокси-объектами, которую вы решили в предыдущем разделе. Вы должны определить момент создания специальной коллекции Hibernate, когда конкретное свойство будет прочитано из документа XML. Для создания этой особой коллекции придется обратиться к внутренним методам Hibernate. Мы советуем использовать коллекции с доступом только на чтение; фактически отображение коллекции – это ссылка на операцию вставки результатов запроса при отправке данных клиенту. Когда клиент отправляет серверу документ XML, он не должен включать элемент `<bids>`. На сервере вы обращаетесь к коллекции хранимого экземпляра `Item` после операции слияния (игнорируя коллекцию во всемя слияния).

Во-вторых, вам, наверное, интересно, где найти примеры с JSON. Мы знаем, что сейчас вы, скорее всего, используете в своих приложениях JSON, а не XML. Формат JSON удобен для анализа клиентом JavaScript. К сожалению, мы не смогли настроить механизм преобразования JSON в JAX-RS без использования нестандартных фреймворков. Хотя JAX-RS может быть стандартизован, но его процедура создания и чтения JSON пока не стандартизована: некоторые реализации JAX-RS используют Jackson, другие выбирают Jettison. Кроме того, появился новый интерфейс Java для обработки JSON (JSONP), который в будущем может быть включен в некоторые реализации JAX-RS.

Чтобы использовать JSON с Hibernate, вам потребуется написать такое же расширение, как мы сделали для JAXB, но только для своего любимого инструмента преобразования JSON. Вы должны будете настроить обработку прокси-объектов, способ отправки данных прокси-объекта клиенту и способ превращения прокси-объекта в ссылку на сущность с помощью `em.getReference()`. Очевидно, вам придется использовать какой-то API для расширения вашего фреймворка, как мы поступили в случае с JAXB.

19.5. Резюме

- Вы увидели несколько способов интеграции Hibernate и JPA в среде веб-приложений. Использовали EntityManager для внедрения в CDI и добавили в уровень хранения обобщенные средства сортировки и постраничной выборки результатов запросов с помощью CDI.
- Рассмотрели применение JPA в приложениях JSF; узнали, как создавать службы с областью видимости запросов и диалоговых взаимодействий, используя JPA в контексте хранения.
- Мы обсудили проблемы, связанные с сериализацией данных, и способы их решения в среде с клиентом без состояния и сервером JAX-RS.

Глава 20

Масштабирование Hibernate

В этой главе:
- выполнение массовых и пакетных операций;
- улучшение масштабирования с помощью общего кэша.

Объектно-реляционное отображение помогает загрузить данные в приложение, чтобы в дальнейшем использовать объектно-ориентированный язык программирования для их обработки. Эта стратегия хорошо подходит для реализации многопользовательского приложения, работающего с маленьким или средним набором данных в каждой отдельной единице работы.

С другой стороны, операции, требующие обработки большого объема данных, не очень подходят для уровня приложения. Такие операции следует размещать как можно ближе к данным, а не наоборот. В SQL-системах для реализации операций, затрагивающих тысячи строк, обычно хватает выражений `UPDATE` и `DELETE`, выполняемых непосредственно в базе данных. Более сложные операции могут потребовать выполнения дополнительных процедур внутри базы данных; следовательно, хранимые процедуры можно рассматривать как одну из возможных стратегий. В приложении с Hibernate в любой момент можно вернуться обратно к JDBC и SQL. Мы уже обсудили некоторые варианты в главе 17. В этой главе мы покажем, как избежать возврата к JDBC и как выполнять массовые и пакетные операции с помощью Hibernate и JPA.

Главным основанием для заявления, что приложения, использующие объектно-реляционный уровень хранения с объектно-реляционным отображением, должны выигрывать в производительности у обычных приложений JDBC, является кэширование. Хотя мы абсолютно уверены, что большинство приложений должно проектироваться так, чтобы иметь приемлемую производительность без кэширования, без сомнения, найдутся такие типы приложений (особенно те, что в основном читают данные или хранят в базе значительный объем метаданных), для которых кэширование окажет значительное влияние на производительность. Более того, масштабирование приложений с высоким уровнем конкурентного доступа, для

поддержки тысяч транзакций в секунду, как правило, требует применения какого-либо механизма кэширования для снижения нагрузки на сервер базы данных. После рассмотрения массовых и пакетных операций мы обсудим систему кэширования Hibernate.

Главные нововведения в JPA 2

- Операции массового удаления и изменения, которые преобразуются в SQL-выражения `DELETE` и `UPDATE`, теперь стандартизованы и доступны в запросах на основе критериев, в интерфейсах JPQL и SQL.
- Параметры конфигурации и аннотации для настройки общего кэша сущностей теперь стандартизированы.

20.1. Массовые и пакетные операции обработки данных

Сначала мы рассмотрим стандартизированные массовые операции в JPQL, такие как `UPDATE` и `DELETE`, а также их эквиваленты в запросах на основе критериев. После этого реализуем некоторые из этих операций с помощью обычных выражений SQL. Затем вы узнаете, как добавлять и изменять большое количество экземпляров сущностей, разделяя их на порции (пакеты). Наконец, мы покажем специальный интерфейс `org.hibernate.StatelessSession`.

20.1.1. Массовые операции в запросах на основе критериев и JPQL

Язык JPQL (язык запросов механизма Java Persistence) очень похож на SQL. Основное различие заключается в том, что вместо имен таблиц JPQL использует имена классов и вместо имен столбцов – имена свойств. JPQL также может работать с наследованием, т. е. дает возможность использовать в запросах суперклассы и интерфейсы. Механизм запросов на основе критериев JPA поддерживает те же конструкции, что и JPQL, дополнительно давая возможность программного создания типизированных выражений.

Следующие выражения, которые мы вам покажем, поддерживают изменение и удаление прямо в базе данных, без необходимости загрузки данных в память. Мы также познакомим вас с выражением, которое может выбирать данные и на этой основе далать вставку новых сущностей прямо на стороне базы данных.

Изменение и удаление экземпляров сущностей

JPA поддерживает операции управления данными, не уступающие по возможностям своим аналогам в SQL. Давайте посмотрим на первую JPQL-операцию: `UPDATE`.

Листинг 20.1 ❖ Выполнение JPQL-выражения `UPDATE`

```
Query query = em.createQuery(
    "update Item i set i.active = true where i.seller = :s"
).setParameter("s", johndoe);

int updatedEntities = query.executeUpdate();

assertEquals(updatedEntities, 2);  ← Экземпляры сущностей, а не «записи»
```

Выражения JPQL выглядят как выражения SQL, но используют имена сущностей (классов) и свойств. Поскольку псевдонимы необязательны, можно написать `update Item set active = true`. Для связывания именованных и позиционных параметров можно использовать стандартный API запросов. Метод `executeUpdate` вернет количество измененных экземпляров сущностей, которое может отличаться от количества измененных записей в базе данных, в зависимости от стратегии отображения.

Выражение `UPDATE` выполняется на стороне базы данных; Hibernate не изменит никаких экземпляров `Item`, загруженных в текущий контекст хранения. В предыдущих главах мы многократно повторяли, что беспокоиться следует только об управлении состоянием экземпляров сущностей, а не о выражениях SQL. Такая стратегия подразумевает, что экземпляры сущностей, на которые вы ссылаетесь, находятся в памяти. Если информацию, загруженную в память приложения, изменить или удалить прямо в базе данных, она не будет изменена или удалена в контексте хранения.

Прагматичное решение такой проблемы заключается в простом соглашении: выполняйте все операции в свежем контексте хранения. Затем вызовите `EntityManager` для загрузки и сохранения экземпляров. Такое соглашение гарантирует, что никакие ранее выполненные выражения не повлияют на контекст хранения. С другой стороны, можно выборочно использовать метод `refresh()` для повторной загрузки экземпляров сущностей в контекст хранения из базы данных, если известно, что данные были изменены вне контекста.

Выражения JPQL/критерии для массовых операций и кэш второго уровня

Выполнение операций прямо в базе данных автоматически очищает необязательный кэш второго уровня в Hibernate. Hibernate анализирует массовые операции JPQL или запросов на основе критериев, определяя, какие области кэша будут затронуты. Затем очищает эти области в кэше второго уровня. Обратите внимание, что этот анализ не отличается детальностью: даже если изменить или удалить лишь несколько строк таблицы `ITEM`, Hibernate очистит *все* области кэша, где хранятся данные класса `Item`.

Ниже показан такой же запрос, но описанный с помощью API запросов на основе критериев:

```
CriteriaUpdate<Item> update =
    criteriaBuilder.createCriteriaUpdate(Item.class);
Root<Item> i = update.from(Item.class);
update.set(i.get(Item_.active), true);
update.where(
    criteriaBuilder.equal(i.get(Item_.seller), johndoe)
);

int updatedEntities = em.createQuery(update).executeUpdate();
```

Еще одно преимущество заключается в том, что JPQL-выражение `UPDATE` и объект `CriteriaUpdate` могут работать с иерархиями наследования. Следующее выражение отмечает все кредитные карты как украденные, если имя пользователя начинается с «J»:

```
Query query = em.createQuery(   ← Для этого изменения Hibernate даже создаст временную таблицу
    "update CreditCard c set c.stolenOn = :now where c.owner like 'J%'"
).setParameter("now", new Date());
```

Hibernate знает, как выполнить это изменение, даже если потребуется сгенерировать несколько выражений SQL или скопировать какие-либо данные во временную таблицу; он изменит строки в нескольких таблицах (поскольку сущность `CreditCard` отображается в несколько таблиц суперкласса и подклассов).

JPQL-выражение `UPDATE` может ссылаться только на один класс сущности, а массовая операция в запросе на основе критериев должна иметь лишь один корневой объект; к примеру, нельзя изменить данные `Item` и `CreditCard` одновременно. В предложении `WHERE` можно использовать подзапросы; любые соединения разрешается использовать только там.

Вы можете изменять значения встроенных типов: например, `update User u set u.homeAddress.street = ...`. Но *нельзя* изменять значения встраиваемых типов в коллекциях. Так делать нельзя: `update Item i set i.images.title = ...`.

Особенности Hibernate

По умолчанию операции не меняют никаких версий или меток времени в модифицируемых сущностях (согласно стандарту JPA). Но расширение Hibernate позволяет увеличить номера версий модифицируемых экземпляров сущностей:

```
int updatedEntities =
   em.createQuery("update versioned Item i set i.active = true")
      .executeUpdate();
```

Версия каждого обновляемого экземпляра сущности `Item` теперь будет увеличена прямо в базе данных, сообщая всем остальным транзакциям, использующим оптимистическое управление параллельным доступом, что данные изменились. (Hibernate не позволит использовать ключевое слово `versioned`, если поле с версией или меткой времени имеет тип `org.hibernate.usertype.UserVersionType`.)

В API запросов на основе критериев JPA версию придется менять вручную:

```
CriteriaUpdate<Item> update =
    criteriaBuilder.createCriteriaUpdate(Item.class);

Root<Item> i = update.from(Item.class);

update.set(i.get(Item_.active), true);
update.set(
    i.get(Item_.version),
    criteriaBuilder.sum(i.get(Item_.version), 1)
);

int updatedEntities = em.createQuery(update).executeUpdate();
```

Следующей массовой операцией, которую мы рассмотрим, будет операция DELETE:

```
em.createQuery("delete CreditCard c where c.owner like 'J%'")
    .executeUpdate();
CriteriaDelete<CreditCard> delete =
    criteriaBuilder.createCriteriaDelete(CreditCard.class);

Root<CreditCard> c = delete.from(CreditCard.class);

delete.where(
    criteriaBuilder.like(
        c.get(CreditCard_.owner),
        "J%"
    )
);

em.createQuery(delete).executeUpdate();
```

К DELETE и CriteriaDelete применяются те же правила: никаких соединений, единственный корневой класс, необязательные псевдонимы, а подзапросы допускаются только в предложении WHERE.

Существует еще одна массовая операция JPQL, позволяющая создавать экземпляры сущностей прямо в базе данных.

Особенности Hibernate

Создание новых экземпляров сущностей

Предположим, что какие-то кредитные карты ваших клиентов были похищены. Вы должны выполнить массовую операцию, чтобы отметить день, когда они были похищены (скорее, день, когда это стало известно), и удалить скомпрометированные данные из ваших записей. Поскольку вы работаете в добросовестной компании, то должны сообщить о происшествии властям и пострадавшим клиентам. Следовательно, прежде чем удалять записи, вам нужно извлечь всю информацию о похищенных данных и создать сотни (а то и тысячи) объектов StolenCreditCard. Специально для этих целей вы создадите отображение класса сущности:

```
@Entity
public class StolenCreditCard {

    @Id
    public Long id;
    public String owner;
    public String cardNumber;
    public String expMonth;
    public String expYear;
    public Long userId;
    public String username;
    public StolenCreditCard() {
    }

    public StolenCreditCard(Long id,
                            String owner, String cardNumber,
                            String expMonth, String expYear,
                            Long userId, String username) {
    }
}
```

Hibernate отобразит этот класс в таблицу `STOLENCREDITCARD`. Затем нужно прямо в базе выполнить выражение, извлекающее данные скомпрометированных кредитных карт и создающее новые объекты `StolenCreditCard`. Это можно сделать с помощью выражения `INSERT ... SELECT`, доступного только в Hibernate:

```
int createdRecords =
    em.createQuery(
        "insert into" +
            " StolenCreditCard(id, owner, cardNumber, expMonth, expYear,
    ➥userId, username)" +
            " select c.id, c.owner, c.cardNumber, c.expMonth, c.expYear, u.id,
    ➥u.username" +
            " from CreditCard c join c.user u where c.owner like 'J%'"
    ).executeUpdate();
```

Эта операция, во-первых, выбирает данные объектов `CreditCard`, принадлежащие соответствующему пользователю (`User`). Во-вторых, вставляет отображаемый класс `StolenCreditCard` прямо в таблицу.

Обратите внимание на следующее:

- свойства, принадлежащие целевому классу выражения `INSERT ... SELECT` (в данном случае это `StolenCreditCard`), должны принадлежать конкретному подклассу, но не (абстрактному) суперклассу. Поскольку класс `StolenCreditCard` не является частью иерархии наследования, здесь это неважно;
- типы, возвращаемые проекцией в предложении `SELECT`, должны соответствовать типам аргументов в предложении `INSERT`;
- в данном примере свойство идентификатора сущности `StolenCreditCard` находится в списке вставляемых свойств и извлекается с помощью выборки; его значение будет совпадать с оригинальным идентификатором сущности

`CreditCard`. С другой стороны, можно отобразить генератор идентификатора класса `StolenCreditCard`; но это подходит только для генераторов, работающих непосредственно на стороне базы данных, таких как последовательности или поля идентичности;

- если для целевого класса настроено версионирование (с помощью поля версии или метки времени), также будет сгенерировано новое значение версии (ноль или текущий момент времени). Также можно выбрать номер версии (или метку времени) и добавить свойство версии (или метки времени) в список вставляемых свойств.

На момент написания этой книги выражение `INSERT ... SELECT` не поддерживалось ни в JPA, ни в API запросов на основе критериев Hibernate.

Массовые операции JPQL и запросов на основе критериев покрывают большинство ситуаций, в которых часто используется обычный SQL. В некоторых случаях может понадобиться выполнить массовые операции SQL, не обращаясь при этом к JDBC.

20.1.2. Массовые операции в SQL

В предыдущем разделе вы видели JPQL-выражения `UPDATE` и `DELETE`. Главное их преимущество заключается в том, что они используют имена классов и свойств, а также в том, что Hibernate умеет работать с иерархиями наследования и версионированием при формировании запросов SQL. Поскольку Hibernate выполняет анализ JPQL, он знает, как эффективно выполнять проверку состояния объектов и выталкивать контекст хранения перед выполнением запроса, а также очищать области кэша второго уровня.

Если в JPQL отсутствует нужная функциональность, массовые операции можно выполнить с помощью SQL:

```
Query query = em.createNativeQuery(
    "update ITEM set ACTIVE = true where SELLER_ID = :sellerId"
).setParameter("sellerId", johndoe.getId());

int updatedEntities = query.executeUpdate();  ← Все области кэша второго уровня будут очищены

assertEquals(updatedEntities, 2);             ← Количество измененных записей, а не сущностей
```

Используя массовые операции в JPA, следует помнить о важной особенности: Hibernate *не* выполняет анализа SQL для определения изменяемых таблиц. Это значит, что Hibernate не знает, нужно ли выталкивать контекст хранения перед выполнением запроса. В предыдущем примере Hibernate не знал, что вы изменяли таблицу `ITEM`. Hibernate должен проверить изменение состояния и сохранить *любые* экземпляры сущностей, присутствующие в контексте хранения, перед выполнением запроса; он не может проверить и сбросить только сущности `Item`.

Если вы активировали кэш второго уровня, вам нужно побеспокоиться о следующем (в противном случае вам не о чем волноваться): Hibernate должен под-

держивать кэш второго уровня в актуальном состоянии, чтобы не возвращать устаревших данных, поэтому он очистит *все* области кэша при выполнении SQL-выражения `UPDATE` или `DELETE`. Следовательно, после выполнения операции кэш второго уровня окажется пустым!

Особенности Hibernate

Но вы можете получить более полный контроль над проверкой состояния объектов выталкиванием контекста и очисткой кэша второго уровня, прибегнув к API для запросов SQL в Hibernate:

```
org.hibernate.SQLQuery query =
    em.unwrap(org.hibernate.Session.class).createSQLQuery(
        "update ITEM set ACTIVE = true where SELLER_ID = :sellerId"
    );
query.setParameter("sellerId", johndoe.getId());

query.addSynchronizedEntityClass(Item.class);

int updatedEntities = query.executeUpdate();

assertEquals(updatedEntities, 2);
```

Будут очищены только области кэша второго уровня с данными сущности Item

Количество измененных записей, а не сущностей

Вызывая метод `addSynchronizedEntityClass()`, вы сообщаете Hibernate, какие таблицы будут изменены выражением SQL, и Hibernate очистит только соответствующие области кэша. Hibernate также понимает, что перед выполнением запроса он должен вытолкнуть лишь измененные экземпляры сущностей `Item`.

Порой при выполнении массовых операций нельзя обойти слой приложения. Приходится загружать данные в память приложения, взаимодействуя с объектом `EntityManager` для изменения и удаления, что приводит к необходимости обрабатывать данные порциями (пакетами).

20.1.3. Пакетная обработка данных

Чтобы создать или изменить сотню или тысячу экземпляров сущностей в одной транзакции и одной единице работы, может не хватить памяти. Более того, требуется учитывать время, необходимое для завершения транзакции. Большинство диспетчеров транзакций имеет малое время ожидания, в пределах секунд или минут. Диспетчер транзакций Bitronix, который использовался в примерах в этой книге, по умолчанию устанавливает время ожидания, равное 60 секундам. Если ваша единица работы требует больше времени для завершения, вы должны сначала переопределить этот предел:

```
tx.setTransactionTimeout(300);
```

5 минут

Это метод класса `UserTransaction`. Только будущие транзакции, запущенные в этом потоке выполнения, станут использовать новое значение времени ожидания. Вы должны установить его до вызова метода `begin()`.

Давайте попробуем записать несколько тысяч экземпляров `Item` в базу пакетами.

Запись наборов экземпляров сущностей

Каждый отсоединенный экземпляр сущности, передаваемый в метод `EntityManager#persist()`, помещается в кэш контекста хранения, как объяснялось в разделе 10.2.8. Чтобы предотвратить полное расходование памяти, нужно вызывать методы `flush()` и `clear()`, выталкивая и очищая контекст хранения, после определенного количества вставок, осуществляя тем самым пакетную запись данных.

Листинг 20.2 ❖ Вставка большого количества экземпляров сущностей

```
tx.begin();
EntityManager em = JPA.createEntityManager();

for (int i = 0; i < ONE_HUNDRED_THOUSAND; i++) {   ← ❶ Создание экземпляров
    Item item = new Item(
        // ...
    );
    em.persist(item);

    if (i % 100 == 0) {                            ← ❷ Выполнение выражений INSERT
        em.flush();
        em.clear();
    }
}

tx.commit();
em.close();
```

❶ Создание и сохранение 100 000 экземпляров `Item`.

❷ После 100 операций контекст хранения выталкивается и очищается. Вследствие этого SQL-выражения `INSERT` будут выполнены для 100 экземпляров `Item`; а поскольку они окажутся в отсоединенном состоянии и на них нигде не будет ссылок, механизм сборки мусора JVM сможет повторно использовать занимаемую ими область памяти.

В настройках единицы хранения вы должны будете присвоить параметру `hibernate.jdbc.batch_size` значение, равное размеру пакета, – в данном случае 100. Используя эту настройку, Hibernate будет накапливать выражения `INSERT` на уровне JDBC с помощью метода `PreparedStatement#addBatch()`.

Пакетное выполнение смешанных выражений SQL

Процедура, сохраняющая пакеты из экземпляров различных сущностей, где последовательно идет экземпляр `Item`, потом `User`, а затем снова `Item` и снова `User` и т. д., будет не очень эффективно накапливать выражения на уровне JDBC. Во время выталкивания контекста Hibernate сначала сгенерирует SQL-выражение `insert into ITEM`, затем `insert into USERS`, а потом опять `insert into ITEM` и т. д. Но Hibernate не сможет выполнить большого пакета выражений за один раз, поскольку каждое следующее выражение отлично от предыдущего. Но если в настройках единицы хра-

нения активировать параметр hibernate.order_inserts, Hibernate отсортирует операции перед созданием наборов выражений. После этого он сначала выполнит все выражения INSERT для таблицы ITEM, а затем выражения INSERT для таблицы USERS. Следовательно, Hibernate сможет создать наборы выражений на уровне JDBC.

Если для сущности Item используется кэш второго уровня, для выполнения вставки пакетов данных его нужно обойти: см. раздел 20.2.5.

Серьезную проблему для массовой вставки данных представляет нагрузка на генератор идентификаторов, потому что каждый вызов EntityManager#persist() должен получить новое значение идентификатора. Обычно генератором является последовательность в базе данных, обращение к которой происходит отдельно для каждого экземпляра сущности. Для эффективной реализации процедуры пакетных изменений следует сократить количество обращений к базе данных.

Особенности Hibernate

В разделе 4.2.5 мы советовали применять доступный только в Hibernate генератор enhanced-sequence, поскольку он поддерживает некоторые оптимизации, идеально подходящие для обработки пакетов данных. Во-первых, нужно описать генератор в файле метаданных package-info.java:

```
@org.hibernate.annotations.GenericGenerator(
  name = "ID_GENERATOR_POOLED",
  strategy = "enhanced-sequence",
  parameters = {
     @org.hibernate.annotations.Parameter(
        name = "sequence_name",
        value = "JPWH_SEQUENCE"
     ),
     @org.hibernate.annotations.Parameter(
         name = "increment_size",
         value = "100"
     ),
     @org.hibernate.annotations.Parameter(
         name = "optimizer",
         value = "pooled-lo"
     )
})
```

Теперь в отображаемых классах сущностей можно воспользоваться генератором, применив аннотацию @GeneratedValue.

Если параметру increment_size присвоить значение 100, последовательность будет выдавать значения 100, 200, 300, 400 и т. д. Оптимизатор pooled-lo генерирует промежуточные значения каждый раз, когда вызывается persist() без лишнего обращения к базе данных. Следовательно, если следующее значение, полученное от последовательности, равняется 100, на уровне приложения Hibernate сгенерирует значения идентификаторов 101, 102, 103 и т. д. Как только набор из

100 идентификаторов закончится, база данных вернет следующее значение последовательности, и процедура повторится снова. Следовательно, чтобы получить набор идентификаторов для пакета из 100 операций вставки, нужно обратиться к базе данных лишь один раз. Есть и другие оптимизаторы генераторов, но оптимизатор `pooled-lo` эффективен практически в любых вариантах использования и прост для понимания и настройки.

Однако помните, что шаг увеличения, равный 100, будет оставлять большие промежутки между числовыми идентификаторами, если приложение будет использовать ту же последовательность, но иной алгоритм. Однако не стоит сильно об этом беспокоиться; если генерировать новый идентификатор каждую миллисекунду, все доступные значения исчерпаются за 3 млн лет, а не за 300.

Этот же прием пакетной обработки данных можно использовать для изменения большого количества экземпляров сущностей.

Особенности Hibernate

Изменение наборов экземпляров сущностей

Представьте, что нужно обработать множество экземпляров `Item` и изменения не такие простые, как изменение флага (что вы уже делали ранее с помощью единственного выражения `UPDATE` в JPQL). Предположим также, что по каким-то причинам нельзя создать хранимую процедуру в базе данных (возможно, из-за того, что приложение работает с СУБД, не поддерживающей хранимых процедур). Остается лишь один выход – написать Java-код, загружающий большое количество данных в память и обрабатывающий с помощью процедуры.

Это требует обработки пакетов данных и перемещения по результатам запроса с помощью курсора, что доступно только в Hibernate. Пожалуйста, перечитайте наше объяснение, как работать с курсорами, в разделе 14.3.3 и проверьте, поддерживаются ли курсоры вашей СУБД и драйвером JDBC. Следующий код будет загружать 100 экземпляров `Item` за один раз.

Листинг 20.3 ❖ Изменение большого количества экземпляров сущностей

```
tx.begin();
EntityManager em = JPA.createEntityManager();

org.hibernate.ScrollableResults itemCursor =  ← ❶ Открыть курсор
   em.unwrap(org.hibernate.Session.class)
       .createQuery("select i from Item i")
       .scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

int count = 0;
while (itemCursor.next()) {                   ← ❷ Переместиться по курсору
    Item item = (Item) itemCursor.get(0);     ← ❸ Извлечь экземпляр

    modifyItem(item);

    if (++count % 100 == 0) {                 ← ❹ Вытолкнуть контекст хранения
```

```
            em.flush();
            em.clear();
        }
    }

    itemCursor.close();
    tx.commit();
    em.close();
```

❶ Здесь выполняется запрос JPQL, загружающий все экземпляры `Item` из базы данных. Чтобы не загружать сразу всех результатов в память, открывается курсор.
❷ Управлять курсором и перемещаться по нему можно с помощью `ScrollableResults` API. Каждый вызов метода `next()` выполняет переход к следующей записи в курсоре.
❸ Вызов `get(int i)` загружает в память приложения одну сущность (запись, на которую указывает курсор).
❹ Чтобы не допустить переполнения памяти, перед загрузкой следующих 100 записей нужно вытолкнуть и очистить контекст хранения.

Для лучшей производительности следует присвоить параметру `hibernate.jdbc.batch_size` в настройках единицы хранения значение 100, равное размеру обрабатываемого пакета. На уровне JDBC Hibernate накапливает все выражения `UPDATE`, которые будут выполнены в момент выталкивания контекста. Если для класса сущности не настроено версионирование, по умолчанию Hibernate не будет накапливать изменений на уровне JDBC – некоторые драйверы JDBC возвращают некорректное число записей, измененных выражением `UPDATE` (в частности, такое поведение наблюдается в Oracle). Если вы уверены, что ваш драйвер JDBC корректно поддерживает это поведение, а класс сущности `Item` имеет аннотацию `@Version`, чтобы активировать пакетную обработку данных в JDBC, нужно присвоить параметру `hibernate.jdbc.batch_versioned_data` значение `true`. Если для сущности `Item` используется кэш второго уровня, для вставки (изменения) порций данных его нужно обойти: см. раздел 20.2.5.

Другой способ, позволяющий избежать переполнения памяти в контексте хранения (по сути, отключая его), состоит в применении интерфейса `org.hibernate.StatelessSession`.

Особенности Hibernate

20.1.4. Интерфейс StatelessSession

Контекст хранения является важнейшей частью механизма Hibernate. Без него не получится управлять состоянием сущности, а Hibernate не сможет автоматически отслеживать изменения. Многое другое также окажется невозможным.

Однако в Hibernate имеется альтернативный интерфейс для тех, кто предпочитает работать с базой данных, просто выполняя выражения. Интерфейс `org.hibernate.StatelessSession` ориентирован на выражения и очень похож на обыч-

ный JDBC, за исключением возможности работы с отображаемыми хранимыми классами и совместимости с базами данных. Наибольший интерес представляют методы `insert()`, `update()` и `delete()`, которые немедленно отображаются в соответствующие операции JDBC/SQL.

Давайте напишем такую же процедуру, изменяющую все товары, но с применением этого интерфейса.

Листинг 20.4 ❖ Изменение данных с помощью `StatelessSession`

```
tx.begin();

org.hibernate.SessionFactory sf =                    ← ❶ Создание экземпляра StatelessSession
    JPA.getEntityManagerFactory().unwrap(org.hibernate.SessionFactory.class);
org.hibernate.StatelessSession statelessSession = sf .openStatelessSession();

org.hibernate.ScrollableResults itemCursor =         ← ❷ Загрузка экземпляров Item
    statelessSession
        .createQuery("select i from Item i")
       .scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

while (itemCursor.next()) {                          ← ❸ Получение экземпляра
    Item item = (Item) itemCursor.get(0);

    modifyItem(item);

    statelessSession.update(item);                   ← ❹ Выполнение операции UPDATE
}

itemCursor.close();
tx.commit();
statelessSession.close();
```

❶ Создание экземпляра `StatelessSession` с помощью фабрики `SessionFactory`, которую можно получить из объекта `EntityManagerFactory`.

❷ Здесь выполняется запрос JPQL, загружающий все экземпляры `Item` из базы данных. Вместо загрузки сразу всех результатов в память здесь открывается курсор.

❸ Последовательное перемещение по результатам с помощью курсора и извлечение экземпляра сущности `Item`. Экземпляр находится в отсоединенном состоянии; контекст хранения отсутствует!

❹ Поскольку в отсутствие контекста хранения Hibernate не может автоматически обнаруживать изменения, приходится выполнять SQL-выражение `UPDATE` вручную.

Отключение контекста хранения и применение интерфейса `StatelessSession` влекут за собой серьезные последствия и существенные ограничения (по крайней мере, по сравнению с обычными объектами `EntityManager` и `org.hibernate.Session`):

- экземпляр `StatelessSession` не имеет кэша контекста хранения и не взаимодействует с другими кэшами второго уровня или кэшем запросов. Отсутствует автоматическая проверка изменения состояния объектов или выполнение кода SQL во время подтверждения транзакции. Все действия влекут немедленное выполнение операций SQL;

- никакие изменения в экземплярах сущностей и никакие операции не передаются каскадно к связанным сущностям. Hibernate игнорирует любые настройки каскадирования в отображениях. Вы работаете с экземплярами одного класса сущности;
- отсутствует область гарантированной идентичности объектов. Один и тот же запрос, выполненный дважды в рамках одного экземпляра `StatelessSession`, вернет два отсоединенных объекта, находящихся в разных областях памяти. Это может привести к эффекту зеркалирования данных (data aliasing), когда два экземпляра сущности будут неразличимы, если неправильно реализовать методы `equals()` и `hashCode()` в хранимом классе;
- Hibernate игнорирует любые изменения коллекций, отображаемых в виде связей сущностей (*один ко многим*, *многие ко многим*). В расчет берутся только коллекции базовых или встраиваемых типов. Следовательно, не нужно отображать связей сущностей с помощью коллекций (кроме отношений *многие к одному* или *один к одному*) и работать с отношением только на другой стороне ассоциации. Вместо последовательного обхода отображаемой коллекции лучше выполнить запрос для получения тех же данных;
- Hibernate не оповещает обработчиков событий JPA и не вызывает методов обратного вызова во время выполнения операций с помощью `StatelessSession`. `StatelessSession` обходит любые перехватчики типа `org.hibernate.Interceptor`, поэтому не следует полагаться на базовый механизм событий Hibernate.

Хороших примеров использования `StatelessSession` немного; к этому способу лучше обращаться, когда обработка пакетов данных с помощью обычного экземпляра `EntityManager` становится слишком запутанной.

В следующем разделе мы представим систему общего кэша Hibernate. Кэширование данных на уровне приложения – это дополнительная оптимизация, которую можно применять в сложных многопользовательских приложениях.

20.2. Кэширование данных

В этом разделе мы покажем, как подключать, настраивать и управлять общим кэшем данных в Hibernate. Общий кэш данных *не является* кэшем контекста хранения, который Hibernate никогда не разделяет между потоками выполнения в приложении. Это не оптимально по причинам, указанным в разделе 10.1.2. Мы называем контекст хранения *кэшем первого уровня*. Общий кэш данных – *кэш второго уровня* – является *необязательным*, и хотя JPA стандартизует некоторые параметры настройки и отображения метаданных для общего кэша, каждая реализация использует свое решение для оптимизации. Начнем с обзора архитектуры общего кэша Hibernate.

Особенности Hibernate

20.2.1. Архтектура общего кэша в Hibernate

Кэш хранит представление текущего состояния базы данных на стороне приложения в памяти или на диске сервера приложений. *Кэш* представляет собой локальную копию данных, находящихся между приложением и базой данных. Проще говоря, кэш Hibernate очень похож на словарь с ключами и значениями. Hibernate может хранить в кэше данные в виде пары ключ/значение и находить требуемое значение по ключу.

В Hibernate есть несколько типов общих кэшей. Кэш можно использовать, чтобы лишний раз не обращаться к базе данных, в одном из следующих случаев:

- приложение выполняет поиск экземпляра сущности по идентификатору (первичному ключу); здесь может произойти обращение к *кэшу данных сущностей*. Такая же операция нужна для инициализации прокси-объекта сущности, поэтому она так же может обратиться к кэшу сущностей, а не к базе данных. В роли ключа выступает значение идентификатора экземпляра сущности, а в роли значения – данные экземпляра сущности (значения его свойств). Эти данные хранятся в разобранном виде, и во время чтения из кэша Hibernate соберет их в один экземпляр;
- механизм хранения выполняет отложенную инициализацию коллекции; *кэш коллекций* может хранить соответствующие элементы. В качестве ключа будет выступать роль коллекции: например, «Item[1234]#bids» будет представлять коллекцию `bids` экземпляра `Item` с идентификатором `1234`. Значением в данном случае будет множество идентификаторов объектов `Bid`, находящихся в коллекции (обратите внимание, что кэш коллекций *не* хранит данных сущностей `Bid`, а только их идентификаторы!);
- приложение выполняет поиск экземпляра сущности по значению уникального атрибута. Это особый *кэш естественных идентификаторов* для классов сущностей с уникальными полями, такими как `User#username`. В роли ключа выступает уникальное поле, такое как `username`, а в роли значения – идентификатор сущности `User`;
- приложение выполняет запрос JPQL, SQL или запрос на основе критериев, а результат итогового запроса SQL уже находится в *кэше результатов запроса*. В роли ключа выступает сам запрос SQL, включающий значения всех параметров, а в роли значения – некоторое представление результата запроса SQL, которое может содержать значения идентификаторов сущностей.

Важно понимать, что кэш данных сущностей – это единственный кэш, где хранятся фактические данные. Три других кэша хранят лишь идентификаторы сущностей. Следовательно, без подключения кэша данных сущностей нет смысла, например, подключать кэш естественных идентификаторов. За удачным поиском в кэше естественных идентификаторов всегда следует обращение к кэшу данных сущностей. Мы подробнее разберем это поведение далее на конкретных примерах.

Как уже упоминалось, кэш в Hibernate имеет двухуровневую организацию.

Хранилище ссылок на неизменяемые данные

Hibernate сохраняет данные в кэше второго уровня в виде разобранной копии, которую он затем собирает во время чтения из кэша. Копирование данных – дорогостоящая операция; поэтому для оптимизации Hibernate может хранить сами неизменяемые данные, не создавая копии в кэше второго уровня. Это очень подходит для ссылочных данных. Предположим, есть класс сущности `City` с полями `zipcode` и `name`, отмеченный как неизменяемый с помощью аннотации `@Immutable`. Если активировать параметр конфигурации единицы хранения `hibernate.cache.use_reference_entries`, Hibernate будет пытаться (за исключением некоторых особых случаев) сохранить ссылку на объект `City` прямо в кэше данных второго уровня. Но если в приложении случайно изменить экземпляр `City`, это изменение коснется всех пользователей (локальной) области кэша, поскольку все будут иметь одну и ту же ссылку.

Кэш второго уровня

На рис. 20.1 показаны различные элементы системы кэширования в Hibernate. В роли кэша первого уровня выступает контекст хранения, как объяснялось в разделе 10.1.2, Hibernate *не* разделяет этот кэш между потоками выполнения; каждый поток имеет свою копию данных в этом кэше. Следовательно, при обращении к этому кэшу не будет *никаких* проблем с изоляцией транзакций и многопоточностью.

Система кэша второго уровня в Hibernate может быть ограничена одним процессом JVM, а может быть распределена по кластеру JVM. Многочисленные потоки приложения могут обращаться к кэшу второго уровня одновременно. *Стратегия многопоточного кэширования* определяет настройку изоляции транзакций для кэша данных сущностей, кэша элементов коллекции и кэша естественных идентификаторов. Во время загрузки или сохранения записи в одном из этих кэшей Hibernate согласует операцию доступа с одной из выбранных стратегий. Выбрать правильную стратегию многопоточного кэширования классов сущностей может быть непросто, но немного позже мы покажем, как это делается, на нескольких примерах.

Кэш результатов запроса имеет свою внутреннюю стратегию поддержки параллельного доступа и актуальности данных. Мы покажем, как работает кэш результатов запросов и для каких запросов его применение имеет смысл.

Реализация механизма кэширования имеет вид подключаемого модуля. В настоящий момент Hibernate вынуждает выбрать одну реализацию механизма кэширования для всей единицы хранения. Реализация механизма кэширования отвечает за управление физическими областями кэширования – ячейками, хранящими данные в слое приложения (в памяти, в индексных файлах или даже в репликах на кластере). Реализация механизма кэширования управляет временем жизни записей и удаляет их по его истечении, сохраняя только самые последние изме-

ненные данные, когда кэш уже переполнен. Реализация поставщика механизма кэширования может взаимодействовать с другими своими экземплярами в кластере JVM для синхронизации данных в ячейках каждого экземпляра. Сам Hibernate не управляет кластером кэшей; это берет на себя механизм кэширования.

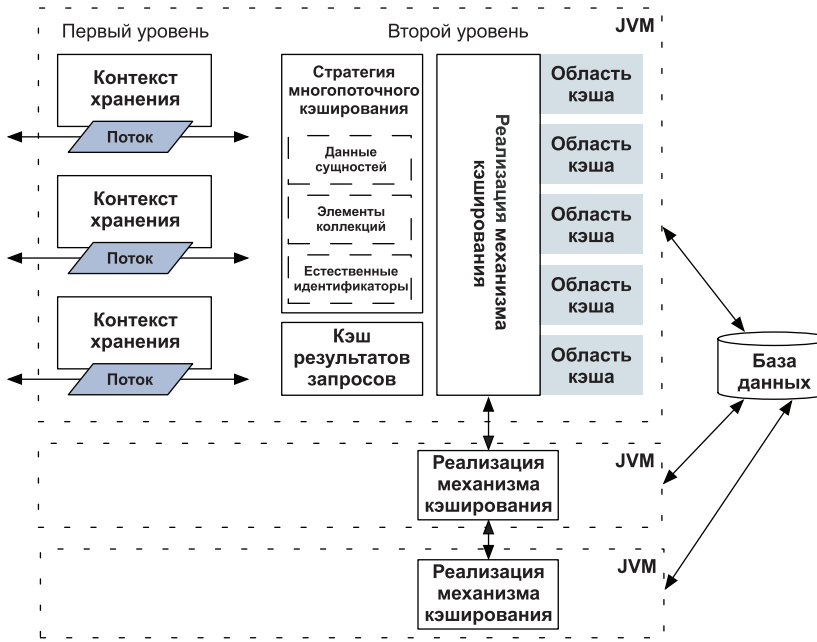


Рис. 20.1 ❖ Двухуровневая организация кэша в Hibernate

В этом разделе мы добавим кэширование для одной виртуальной машины JVM, используя реализацию *Ehcache* – простой, но мощный механизм (изначально разработанный для Hibernate в качестве *простого кэша Hibernate*). Мы рассмотрим лишь некоторые из базовых настроек Ehcache; за более подробной информацией обращайтесь к руководству.

Чаще всего первый вопрос, который задают разработчики: «Будет ли кэш знать, что данные изменились в базе?» Давайте попробуем ответить на него прежде, чем приступим к настройке и использованию кэша.

Кэширование и многопоточность

Если приложение *не имеет* исключительного доступа к базе данных, применение общего кэша целесообразно лишь для редко изменяющихся данных, когда небольшой период рассогласования некритичен. Если другое приложение изменит данные в базе, ваш кэш будет хранить устаревшую версию, пока срок ее жизни не подойдет к концу. Другим приложением может оказаться хранимая процедура, вы-

зываемая триггером, или даже настройка внешнего ключа `ON DELETE` или `ON UPDATE`. Кэш в Hibernate просто не имеет возможности узнать, что другое приложение или триггер базы данных что-то изменили; база данных не может послать сообщения об этом. (Вы можете реализовать такую возможность с помощью триггеров базы данных и JMS, и сделать это довольно просто.) Следовательно, использование кэша зависит от типа данных и требований к их актуальности для конкретного варианта использования.

Предположим на минуту, что приложение имеет исключительный доступ к базе данных. Но даже в этом случае вы должны задать себе те же вопросы, поскольку данные, загружаемые в общий кэш из базы данных в одной транзакции, могут стать видимы для другой транзакции. Какую изоляцию транзакций должен обеспечивать механизм кэширования? Наличие общего кэша повлияет на уровень изоляции ваших транзакций независимо от того, читаете вы только подтвержденные данные или используете повторяющееся чтение. Для некоторых данных может быть допустимо, чтобы обновления в одном потоке приложения не становились сразу видимы другим потокам, создавая приемлемый интервал рассогласования. Это позволит применить более эффективную и агрессивную стратегию кэширования.

Начните проектирование с рассмотрения диаграммы моделей предметной области и классов сущностей. Для кэширования отлично подойдут классы, представляющие:

- редко изменяющиеся данные;
- некритичные данные (например, данные управления контентом);
- локальные данные приложения, которые не могут быть изменены другими приложениями.

Плохими кандидатами являются:

- часто изменяемые данные;
- финансовые данные, когда решения принимаются на основе последних изменений;
- данные, разделяемые с другими приложениями, которые могут изменяться.

Но это не все правила, которые мы обычно применяем. В большинстве приложений есть классы со следующими свойствами:

- небольшое количество экземпляров (тысячи, но не миллионы), легко умещающееся в памяти;
- на каждый экземпляр класса есть множество ссылок из экземпляров других классов или
- экземпляров классов, изменяемых редко (или никогда).

Мы называем такой тип данных *ссылочными данными*. В качестве примера можно привести классы представления почтовых кодов, адресов, статического текста сообщений и т. д. Ссылочные данные идеально подходят для хранения в общем кэше, а приложения, использующие такие данные, могут получить большие преимущества от их кэширования. Данные должны обновляться по истечении определенного времени, а после обновления будет небольшой, но приемлемый период

рассогласования. Фактически для некоторых ссылочных данных (таких как коды стран) допустим очень большой период рассогласования, и их можно вообще не обновлять, если они доступны только для чтения.

Вы должны тщательно анализировать каждый класс или коллекцию, для которой хотите использовать кэширование. Также вы должны выбрать правильную стратегию многопоточного кэширования доступа.

Выбор стратегии многопоточного кэширования

Реализация стратегии многопоточного кэширования является посредником: она отвечает за сохранение данных в кэше и их последующее извлечение из него. Эта важная роль определяет семантику изоляции транзакций для конкретного объекта. Для каждого хранимого класса или коллекции придется выбрать, какую стратегию многопоточного кэширования использовать в случае использования общего кэша.

В Hibernate поддерживаются четыре стратегии многопоточного кэширования, соответствующие убывающим уровням строгости в терминах изоляции транзакций:

- `TRANSACTIONAL` – доступная только в средах с диспетчером системных транзакций, эта стратегия гарантирует полную изоляцию транзакций вплоть до уровня *повторяемого чтения*, если такой уровень поддерживается механизмом кэширования. При использовании этой стратегии Hibernate предполагает, что механизм кэширования знает о системных транзакциях и принимает в них участие. Hibernate не предпринимает никаких попыток блокировать или проверить версии; изоляцию данных в параллельных транзакциях он целиком доверяет механизму кэширования. Эту стратегию лучше использовать для данных, которые в основном читаются, когда важно предотвратить чтение их устаревших значений в параллельных транзакциях и когда изменения происходят достаточно редко. Такая стратегия будет работать и в кластере, если механизм кэширования поддерживает синхронное распределенное кэширование;
- `READ_WRITE` – поддерживает уровень изоляции с *чтением подтвержденных данных*, когда Hibernate может использовать метки времени; такая стратегия уже не может применяться в кластерах. Hibernate также может использовать для блокировки нестандартный API механизма кэширования. Используйте эту стратегию для данных, которые обычно только читаются, когда важно предотвратить чтение устаревших значений в параллельных транзакциях, а изменения редки. Эту стратегию не стоит использовать, когда данные параллельно меняются на стороне базы данных;
- `NONSTRICT_READ_WRITE` – не дает никаких гарантий согласованности между кэшем и базой данных. Транзакция может извлечь из кэша устаревшие данные. Используйте эту стратегию, если данные практически не меняются (скажем, не каждые 10 секунд), а период рассогласования не сильно важен. Период рассогласования настраивается с помощью политики обновления

данных в механизме кэширования. Такая стратегия может применяться и в кластере, даже если он использует асинхронное распределенное кэширование. Другие приложения вполне могут менять те же данные в базе;

- `READ_ONLY` – подходит для неизменяемых данных. Вы получите исключение при попытке изменения. Используйте ее только для ссылочных данных.

По мере уменьшения строгости увеличиваются производительность и возможность масштабирования. Асинхронный кэш кластера со стратегией `NONSTRICT_READ_WRITE` сможет поддерживать куда больше транзакций, чем синхронный со стратегией `TRANSACTIONAL`. Прежде чем использовать полную изоляцию транзакций в рабочем окружении, тщательно исследуйте производительность кэша кластера с ее применением. Если ни в коем случае нельзя получать устаревшие данные, в большинстве случаев стоит отключить распределенное кэширование для конкретного класса!

Начните измерять производительность приложения с отключенным общим кэшем. Затем подключайте его для подходящих классов, постоянно тестируя масштабируемость системы, оценивая различные стратегии многопоточного кэширования. Для оценки влияния изменений на настройку кэша вам понадобятся автоматические тесты. Мы советуем заранее написать тесты на производительность и масштабирование критических участков приложения до подключения разделяемого кэша.

Теперь, зная теорию, пришло время узнать, как кэширование работает на практике.

Сначала настроим общий кэш.

20.2.2. Настройка общего кэша

Общий кэш настраивается в файле конфигурации persistence.xml.

Листинг 20.5 ❖ Настройка общего кэша в файле persistence.xml

Файл: /model/src/main/resources/META-INF/persistence.xml

```
<persistence-unit name="CachePU">
    ...
    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>  ← ❶ Режим общего кэша
    <properties>
        <property name="hibernate.cache.use_second_level_cache"
                  value="true"/>   ← ❷ Подключение кэша второго уровня
        <property name="hibernate.cache.use_query_cache"
                  value="true"/>
        <property name="hibernate.cache.region.factory_class"
                  value="org.hibernate.cache.ehcache.   ← ❸ Механизм кэша второго уровня
                        ➥SingletonEhCacheRegionFactory"/>

        <property name="net.sf.ehcache.configurationResourceName"
                  value="/cache/ehcache.xml"/>  ← ❹ Местоположение файла конфигурации
        <property name="hibernate.cache.use_structured_entries"
                  value="false"/>  ← ❺ Управление сборкой экземпляров сущностей
```

```
        <property name="hibernate.generate_statistics"
                  value="true"/>    ← ❻ Сбор статистики
    </properties>
</persistence-unit>
```

❶ Режим общего кэша влияет на способ активации кэширования для классов сущностей в единице хранения. Обычно кэш подключается избирательно, лишь для некоторых классов сущностей. Варианты: `DISABLE_SELECTIVE`, `ALL` и `NONE`.

❷ Кэш второго уровня в Hibernate нужно подключать явно; по умолчанию он отключен. Кэш результатов запросов также подключается отдельно; по умолчанию он отключен.

❸ Выбор реализации кэша второго уровня. Для использования Ehcache добавьте зависимость от Maven-артефакта `org.hibernate:hibernate-ehcache` в путь к классам. Затем в настройках фабрики для данной области кэширования выберите способ взаимодействия Hibernate с Ehcache; здесь Hibernate будет использовать в качестве механизма кэширования второго уровня единственный экземпляр Ehcache.

❹ Hibernate передаст местоположение файла конфигурации механизму Ehcache во время запуска. В этом файле находятся все настройки физических областей кэширования.

❺ Эта настройка управляет сборкой и разборкой экземпляров сущностей во время загрузки и сохранения в кэше второго уровня. Структурированный формат записей кэша менее эффективен, но необходим для работы в условиях кластера. Для кэша второго уровня, работающего вне кластера, такого как объект-одиночка Ehcache в данной виртуальной машине JVM, можно отключить эту настройку для применения более эффективного формата.

❻ Экспериментируя с кэшем второго уровня, обычно требуется знать, что происходит внутри. В Hibernate есть механизм сбора статистики и средства ее извлечения. Из соображений производительности он по умолчанию выключен (и должен оставаться таким в рабочем окружении).

Теперь кэш второго уровня настроен, и Hibernate запустит Ehcache во время создания объекта `EntityManagerFactory` для данной единицы хранения. По умолчанию Hibernate ничего не кэширует; для классов сущностей и их коллекций нужно настраивать кэширование избирательно.

20.2.3. Кэширование коллекций и сущностей

Сейчас мы рассмотрим классы сущностей и коллекции из предметной модели приложения CaveatEmptor и настроим кэширование, выбрав оптимальную стратегию. Параллельно настроим необходимые физические области кэша в файле конфигурации Ehcache.

Начнем с сущности `User`: эти данные меняются сравнительно редко, тем не менее пользователь может периодически менять свое имя или адрес. Это не важные данные в финансовом отношении; немногие станут принимать решения о покупке на основании имени пользователя или его адреса. Небольшой период рассогласования вполне допустим, если пользователь поменяет имя или адрес. Предположим, что если в течение максимум одной минуты другие транзакции будут видеть устаревшую информацию, никаких проблем не возникнет. Это значит, что можно применить стратегию кэширования `NONSTRICT_READ_WRITE`:

Файл: /model/src/main/java/org/jpwh/model/cache/User.java

```
@Entity
@Table(name = "USERS")
@Cacheable
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations
              .CacheConcurrencyStrategy.NONSTRICT_READ_WRITE,
    region = "org.jpwh.model.cache.User"   ← Имя по умолчанию
)
@org.hibernate.annotations.NaturalIdCache
public class User {

    @NotNull                    ← Не используется при генерации схемы из-за аннотации @NaturalId
    @org.hibernate.annotations.NaturalId(mutable = true)
    @Column(nullable = false)   ← Для генерации схемы
    protected String username;

    // ...
}
```

Добавляет ограничение уникальности UNIQUE

Особенности Hibernate

Аннотация `@Cacheable` активирует общий кэш для этого класса сущности, но для выбора стратегии нужно использовать аннотации Hibernate. Hibernate будет сохранять данные из сущности `User` в кэше второго уровня, в области с именем `your.packag.name.User`. Переопределить имя можно с помощью атрибута `region` аннотации `@Cache`. (Также можно задать глобальный префикс для имени области с помощью параметра `hibernate.cache.region_prefix` в настройках единицы хранения.)

Еще включим для сущности `User` кэш естественных идентификаторов, добавив аннотацию `@org.hibernate.annotations.NaturalIdCache`. Свойства, определяющие естественный идентификатор, отмечены аннотацией `@org.hibernate.annotations.NaturalId`, и вы должны сообщить Hibernate, меняется ли значение каждого поля. Это позволит выполнять поиск экземпляров `User` по полю `username` без обращения к базе данных.

Далее в Ehcache нужно настроить области как для кэша данных сущности, так и для кэша естественных идентификаторов:

Файл: /model/src/main/resources/cache/ehcache.xml

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">

    <cache name="org.jpwh.model.cache.User"
           maxElementsInMemory="500"
           eternal="false"
           timeToIdleSeconds="30"
           timeToLiveSeconds="60"/>

    <cache name="org.jpwh.model.cache.User##NaturalId"
```

```
        maxElementsInMemory="500"
        eternal="false"
        timeToIdleSeconds="30"
        timeToLiveSeconds="60"/>

</ehcache>
```

В каждом из кэшей можно хранить не более 500 записей, и Ehcache не будет хранить их постоянно. Ehcache удалит элемент, если к нему не было обращений в течение 30 секунд, а любой активный элемент он удалит через 1 минуту. Это обеспечивает период рассогласования данных при чтении из кэша не более одной минуты. Иначе говоря, области кэша будут содержать 500 последних использовавшихся записей, каждая не старше 1 минуты, и они будут автоматически сокращаться.

Перейдем к классу сущности `Item`. Эти данные меняются часто, но читаются гораздо чаще, чем записываются. Если поменялось название или описание товара, параллельная транзакция должна немедленно увидеть это изменение. Пользователи принимают финансовые решения о покупке товара на основании его описания. Следовательно, подходящей стратегией будет `READ_WRITE`:

Файл: /model/src/main/java/org/jpwh/model/cache/Item.java

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
)
public class Item {

    // ...
}
```

Hibernate будет согласовывать чтение и запись при изменении данных `Item`, гарантируя чтение из общего кэша только подтвержденных данных. Но если другое приложение изменит данные `Item` прямо в базе, ждать можно чего угодно! Область кэша в Ehcache настраивается так, чтобы последние использованные экземпляры `Item` удалялись через час, чтобы не допустить заполнения ячеек кэша устаревшими данными:

Файл: /model/src/main/resources/cache/ehcache.xml

```
<cache name="org.jpwh.model.cache.Item"
       maxElementsInMemory="5000"
       eternal="false"
       timeToIdleSeconds="600"
       timeToLiveSeconds="3600"/>
```

Рассмотрим коллекцию `bids` класса сущности `Item`: конкретный объект `Bid` в коллекции `Item#bids` не меняется, но сама коллекция может, поэтому параллельные единицы работы должны сразу же видеть все добавления и удаления:

Файл: /model/src/main/java/org/jpwh/model/cache/Item.java

```
public class Item {

    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.Cache(
        usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
    )
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Область кэша настраивается с теми же параметрами, что и для класса сущности-владельца, поскольку каждый объект `Item` имеет коллекцию `bids`:

Файл: /model/src/main/resources/cache/ehcache.xml

```
<cache name="org.jpwh.model.cache.Item.bids"
       maxElementsInMemory="5000"
       eternal="false"
       timeToIdleSeconds="600"
       timeToLiveSeconds="3600"/>
```

Но помните, что кэш коллекции *не* хранит данных из объектов `Bid`. В кэше коллекции хранятся только идентификаторы объектов `Bid`. Следовательно, необходимо также настроить кэширование сущностей `Bid`. В противном случае во время обхода коллекции `Item#bids` Hibernate будет сначала обращаться к кэшу, но затем, из-за отсутствия данных, будет читать данные каждого отдельного объекта `Bid` из базы. Это тот случай, когда включение кэширования *увеличит* нагрузку на сервер базы данных!

Поскольку класс `Bid` неизменяемый, для него подойдет стратегия `READ_ONLY`:

Файл: /model/src/main/java/org/jpwh/model/cache/Bid.java

```
@Entity
@org.hibernate.annotations.Immutable
@Cacheable
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY
)
public class Bid {

    // ...
}
```

Хотя экземпляры `Bid` не изменяются, нужно настроить политику обновления области кэша, чтобы устаревшие данные не заполнили кэш целиком:

Файл: /model/src/main/resources/cache/ehcache.xml

```
<cache name="org.jpwh.model.cache.Bid"
       maxElementsInMemory="100000"
```

```
    eternal="false"
    timeToIdleSeconds="600"
    timeToLiveSeconds="3600"/>
```

Теперь мы готовы проверить работу кэша и узнать, как работает кэширование в Hibernate.

20.2.4. Проверка работы разделяемого кэша

Работу внутреннего кэширования в Hibernate бывает трудно проанализировать. Для сохранения и загрузки данных по-прежнему используется `EntityManager`, а Hibernate самостоятельно записывает данные в кэш и читает их из него. Конечно, обращения к базе данных можно увидеть, активировав журналирование SQL-выражений Hibernate, но вам также стоит познакомиться с классом `org.hibernate.stat.Statistics`, помогающим получить больше информации о единице работы и происходящим за кулисами. Давайте на примерах посмотрим, как он работает.

Мы уже настроили сбор статистики ранее, в конфигурации единицы хранения, в разделе 20.2.2. Статистику единицы хранения можно получить с помощью `org.hibernate.SessionFactory`:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
Statistics stats =
    JPA.getEntityManagerFactory()
        .unwrap(SessionFactory.class)
        .getStatistics();

SecondLevelCacheStatistics itemCacheStats =
    stats.getSecondLevelCacheStatistics(Item.class.getName());
assertEquals(itemCacheStats.getElementCountInMemory(), 3);
assertEquals(itemCacheStats.getHitCount(), 0);
```

Здесь мы также получаем статистику области кэша с данными сущностей `Item`, и, как видите, в кэше уже есть несколько записей. Это *теплый* кэш; Hibernate положил данные в кэш, когда приложение сохраняло экземпляры сущностей `Item`. Чтения сущностей из кэша, однако, не происходило, поэтому счетчик успешных чтений равен нулю.

Если теперь выполнить поиск экземпляра `Item` по идентификатору, Hibernate попытается прочитать данные из кэша, не выполняя SQL-выражения `SELECT`:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
Item item = em.find(Item.class, ITEM_ID);
assertEquals(itemCacheStats.getHitCount(), 1);
```

В кэше также имеются данные из сущностей `User`, поэтому при инициализации коллекции `Item#seller` также произойдет чтение из кэша:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
SecondLevelCacheStatistics userCacheStats =
    stats.getSecondLevelCacheStatistics(User.class.getName());
assertEquals(userCacheStats.getElementCountInMemory(), 3);
assertEquals(userCacheStats.getHitCount(), 0);

User seller = item.getSeller();
assertEquals(seller.getUsername(), "johndoe");  ← Инициализация прокси-объекта

assertEquals(userCacheStats.getHitCount(), 1);
```

Hibernate будет использовать кэш также при обходе коллекции `Item#bids`:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
SecondLevelCacheStatistics bidsCacheStats =      ← ❶ Подсчет элементов коллекции Item#bids
    stats.getSecondLevelCacheStatistics(Item.class.getName() + ".bids");
assertEquals(bidsCacheStats.getElementCountInMemory(), 3);
assertEquals(bidsCacheStats.getHitCount(), 0);

SecondLevelCacheStatistics bidCacheStats =       ← ❷ Подсчет элементов Bid
    stats.getSecondLevelCacheStatistics(Bid.class.getName());
assertEquals(bidCacheStats.getElementCountInMemory(), 5);
assertEquals(bidCacheStats.getHitCount(), 0);

Set<Bid> bids = item.getBids();                  ← ❸ Чтение из кэшей
assertEquals(bids.size(), 3);

assertEquals(bidsCacheStats.getHitCount(), 1);   ← ❹ Результаты обращения к кэшам
assertEquals(bidCacheStats.getHitCount(), 3);
```

❶ Статистика говорит о том, что в кэше находятся три коллекции `Item#bids` (по одной для каждого объекта `Item`). Пока к кэшу не было успешных обращений.
❷ В кэше сущностей `Bid` есть пять записей, которые также еще не читались.
❸ Инициализация коллекции влечет чтение из обоих кэшей.
❹ Из кэша была прочитана одна коллекция, а также три ее элемента `Bid`.

Работа специального кэша естественных идентификаторов сущностей `User` *не* полностью скрыта внутри. Для поиска по естественному идентификатору нужно вызвать метод интерфейса `org.hibernate.Session`:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
NaturalIdCacheStatistics userIdStats =           ← ❶ Подсчет объектов User
    stats.getNaturalIdCacheStatistics(User.class.getName() + "##NaturalId");
assertEquals(userIdStats.getElementCountInMemory(), 1);

User user = (User) session.byNaturalId(User.class)  ← ❷ Поиск по естественному идентификатору
    .using("username", "johndoe")
    .load();

assertNotNull(user);
```

```
assertEquals(userIdStats.getHitCount(), 1);          ← ❸ Естественный идентификатор найден

SecondLevelCacheStatistics userStats =               ← ❹ Сущность найдена
    stats.getSecondLevelCacheStatistics(User.class.getName());
assertEquals(userStats.getHitCount(), 1);
```

❶ В области кэша естественных идентификаторов сущности `User` имеется одна запись.
❷ `org.hibernate.Session` выполняет поиск по естественному идентификатору; это единственный API для работы с кэшем естественных идентификаторов.
❸ Поиск по естественному идентификатору завершился успехом. Кэш вернул идентификатор для значения «johndoe».
❹ Поиск данных сущности `User` также оказался успешным.

API для работы со статистикой предлагает гораздо больше информации, чем было показано в этих простых примерах; мы советуем глубже изучить его. Hibernate собирает информацию обо всех операциях, и эта статистика может помочь найти узкие места приложения: например, запросы, выполняющиеся дольше всего, или сущности и коллекции, к которым чаще всего происходит обращение.

Получение статистики с помощью JMX

Статистику Hibernate во время выполнения можно получать, используя стандартное расширение управления Java (Java Management Extension, JMX). Для этого нужно лишь зарегистрировать объект `Statistics` в качестве экземпляра MBean; с помощью динамического прокси-объекта для этого понадобится лишь несколько строк кода. Мы добавили пример в `org.jpwh.test.cache.SecondLevel`.

Как упоминалось в начале этого раздела, Hibernate неявно записывает данные в кэш и читает их из него. В некоторых случаях может понадобиться получить больший контроль над использованием кэша либо явно отказаться от его использования. Здесь в игру вступают режимы кэширования.

20.2.5. Установка режимов кэширования

JPA стандартизует управление общим кэшем с помощью нескольких *режимов кэширования*. Следующий вызов `EntityManager#find()`, к примеру, не ищет данных в кэше, а обращается к базе напрямую:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
Map<String, Object> properties = new HashMap<String, Object>();
properties.put("javax.persistence.cache.retrieveMode",
    CacheRetrieveMode.BYPASS);
Item item = em.find(Item.class, ITEM_ID, properties);  ← Обращение к базе данных
```

По умолчанию используется значение `USE` перечисления `CacheRetrieveMode`, отвечающего за режим чтения из кэша; здесь мы заменили его на `BYPASS` для выполнения отдельной операции.

Но чаще для задания режима кэширования используются значения перечисления `CacheStoreMode`, отвечающего за сохранение в кэш. По умолчанию Hibernate сохраняет данные в кэше при вызове `EntityManager#persist()`. То же самое происходит при извлечении экземпляра сущности из базы. Но если сохраняется или извлекается большое количество экземпляров сущностей, может наступить переполнение кэша. Это особенно важно в случае пакетной обработки данных, показанной ранее в этой главе.

Сохранение данных в разделяемом кэше сущностей можно отключить для всей единицы работы, передав значение перечисления `CacheStoreMode` объекту `EntityManager`:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
em.setProperty("javax.persistence.cache.storeMode", CacheStoreMode.BYPASS);

Item item = new Item(
    // ...
);

em.persist(item);  ← Не сохраняется в кэш
```

Давайте рассмотрим особый режим кэширования – `CacheStoreMode.REFRESH`. При загрузке экземпляра сущности из базы по умолчанию выбирается режим `CacheStoreMode.USE`, и Hibernate сначала пытается найти загружаемый экземпляр сущности в кэше. Затем, если кэш уже содержит то, что нужно, Hibernate не сохранит загруженных данных в кэш. Это позволяет избежать записи в кэш и удешевить операцию чтения из кэша. Работая в режиме `REFRESH`, Hibernate всегда будет сохранять загружаемые данные в кэш, не читая его перед этим.

В кластере с синхронным распределенным кэшированием запись в каждый из узлов кэша часто является довольно дорогостоящей операцией. Фактически, используя распределенный кэш, вы должны установить значение параметра конфигурации `hibernate.cache.use_minimal_puts` в `true`. Это позволяет оптимизировать операции с кэшем второго уровня путем уменьшения количества операций записи, но ценой большего количества операций чтения. С другой стороны, если для вашего механизма кэширования и архитектуры нет разницы между чтением и записью, можно избавиться от дополнительного чтения с помощью режима `CacheStoreMode.REFRESH`. (Обратите внимание, что некоторые механизмы кэширования могут устанавливать значение параметра `use_minimal_puts` самостоятельно: например, в Ehcache эта настройка по умолчанию активна.)

Как вы уже видели, режим кэширования может устанавливаться для метода `find()` или для всего объекта `EntityManager`. Также можно задать режим для операции `refresh()` или для отдельного запроса `Query`, как обсуждалось в разделе 14.5. Режим отдельного запроса или метода переопределяет режим, установленный для `EntityManager`.

Режим кэширования влияет лишь на внутреннюю работу Hibernate с кэшем. Но иногда может понадобиться управлять кэшем программно: например, для удаления данных из него.

20.2.6. Управление разделяемым кэшем

Стандартный инструмент JPA для управления кэшем – интерфейс `Cache`:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
EntityManagerFactory emf = JPA.getEntityManagerFactory();
Cache cache = emf.getCache();

assertTrue(cache.contains(Item.class, ITEM_ID));
cache.evict(Item.class, ITEM_ID);
cache.evict(Item.class);
cache.evictAll();
```

Это простой API, который позволяет обращаться только к областям кэша с данными сущностей. Для доступа к другим областям, таким как области коллекций или естественных идентификаторов, нужно использовать `org.hibernate.Cache`:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
org.hibernate.Cache hibernateCache =
    cache.unwrap(org.hibernate.Cache.class);

assertFalse(hibernateCache.containsEntity(Item.class, ITEM_ID));
hibernateCache.evictEntityRegions();
hibernateCache.evictCollectionRegions();
hibernateCache.evictNaturalIdRegions();
hibernateCache.evictQueryRegions();
```

Этот механизм управления редко бывает полезен. Также обратите внимание, что удаление записей из кэша второго уровня происходит вне транзакции: т. е. Hibernate не блокирует областей кэша во время удаления.

Перейдем к последней части системы кэширования в Hibernate – кэшу результатов запроса.

20.2.7. Кэш результатов запросов

Кэш результатов запроса по умолчанию выключен, и любой запрос на основе критериев, JPA-запрос или обычный запрос SQL обращается непосредственно к базе данных. В этом разделе мы покажем, почему Hibernate по умолчанию не использует кэша результатов запросов и как включить его для конкретных запросов, если потребуется.

Следующая процедура выполняет запрос JPQL, сохраняя результат в особой области кэша результатов запросов:

Файл: /examples/src/test/java/org/jpwh/test/cache/SecondLevel.java

```
String queryString = "select i from Item i where i.name like :n";

Query query = em.createQuery(queryString)      ← ❶ Включение кэширования
    .setParameter("n", "I%")
    .setHint("org.hibernate.cacheable", true);

List<Item> items = query.getResultList();      ← ❷ Выполнение запроса
```

```
assertEquals(items.size(), 3);

QueryStatistics queryStats = stats.getQueryStatistics(queryString);   ← ❸ Получение информации
assertEquals(queryStats.getCacheHitCount(), 0);
assertEquals(queryStats.getCacheMissCount(), 1);
assertEquals(queryStats.getCachePutCount(), 1);

SecondLevelCacheStatistics itemCacheStats =   ← ❹ Сохранение данных в кэше сущностей
    stats.getSecondLevelCacheStatistics(Item.class.getName());
assertEquals(itemCacheStats.getElementCountInMemory(), 3);
```

❶ Нужно включить кэширование для конкретного запроса. Без рекомендации `org.hibernate.cachable` результат не будет помещен в кэш результатов запроса.
❷ Hibernate выполняет запрос SQL, извлекая результат запроса в память.
❸ Нужную информацию можно получить, используя механизм сбора статистики. Поскольку запрос выполняется впервые, кэш обнаружит промах. Hibernate поместит запрос с результатами в кэш. Если выполнить этот запрос снова, результат уже вернется из кэша.
❹ Данные экземпляра сущности, полученные в виде результата запроса, будут сохранены в области кэша сущностей, а не в кэше результатов запроса.

Рекомендация `org.hibernate.cachable` передается `Query` API, поэтому она будет работать и с обычными запросами SQL, и с запросами на основе критериев. Внутри, в качестве ключа кэша Hibernate использует сам запрос SQL, в котором символы подстановки заменены фактическими аргументами.

Кэш результатов запросов не хранит результатов целиком. В последнем примере результаты содержали данные из таблицы `ITEM`. Hibernate игнорирует большую часть информации в этом результате; в кэше сохранятся только значения атрибута `ID` для каждой строки таблицы `ITEM`. Значения полей каждого экземпляра `Item` сохранятся в области кэша данных сущностей.

Если теперь выполнить тот же запрос, подставляя те же значения аргументов для параметров, Hibernate сначала обратится к кэшу результатов запроса. Он извлечет значения идентификаторов записей `ITEM` из области кэша данных запроса. Затем выполнит поиск по идентификатору и сборку экземпляров сущностей `Item`, обращаясь к области кэша данных сущностей. Если вы решили применить кэширование для запросов, извлекающих сущности, не забудьте включить обычное кэширование для этих сущностей. В противном случае вы получите *больше* обращений к базе данных после подключений кэша результатов запроса!

Если кэшировать запросы, возвращающие скалярные или встраиваемые значения, а не экземпляры сущностей (например, `select i.name from Item i` или `select u.homeAddress from User`), эти значения будут храниться прямо в области кэша результатов запроса.

Кэш результатов запроса состоит из двух физических областей:

Файл: /model/src/main/resources/cache/ehcache.xml

```
<cache name="org.hibernate.cache.internal.StandardQueryCache"
       maxElementsInMemory="500"
```

```
        eternal="false"
        timeToIdleSeconds="600"
        timeToLiveSeconds="3600"/>

<cache name="org.hibernate.cache.spi.UpdateTimestampsCache"
        maxElementsInMemory="50"
        eternal="true"/>
```

В первой области хранятся результаты запроса. Вы должны обеспечить удаление записей по истечении определенного времени, чтобы свободное место использовалось для самых последних запросов.

Вторая область, `org.hibernate.cache.spi.UpdateTimestampsCache`, играет особую роль: Hibernate использует ее, чтобы определить, когда устареют результаты запроса в кэше. Если повторно выполнить запрос с активным кэшированием, Hibernate проверит область меток времени, чтобы узнать о самом последнем добавлении, изменении или удалении, сделанном в таблице (таблицах), участвующей в запросе. Если найденная метка времени больше метки последнего сохраненного результата, Hibernate избавится от него и выполнит новый запрос к базе данных. Это гарантирует, что Hibernate не будет использовать результатов кэша запросов, если хотя бы одна из таблиц содержит обновленные данные; следовательно, результат в кэше мог устареть. Чтобы механизм кэширования всегда хранил устаревшие записи в кэше меток времени, нужно отключить их удаление. Максимальное количество записей в этой области кэша определяется числом таблиц в отображаемой модели.

Большинство запросов ничего не выиграет от кэширования. Это может показаться удивительным. В конце концов, уменьшение количества обращений к базе данных – это всегда хорошо. Но есть две причины, из-за которых кэширование произвольных запросов не всегда работает, в отличие от поиска сущности по идентификатору или инициализации коллекции.

Во-первых, нужно понимать, как часто этот запрос будет выполняться с одними и теми же аргументами. Очевидно, что приложение может последовательно выполнять несколько запросов с одинаковыми аргументами, связанными с параметрами, и одинаковым текстом запроса SQL. Мы считаем, такое встречается редко, но если вы уверены, что запрос будет выполняться повторно, он станет отличным кандидатом для кэширования результатов.

Во-вторых, для приложений, выполняющих много запросов, но мало вставок, изменений и удалений, кэширование результатов запросов может улучшить производительность и масштабирование. С другой стороны, если приложение делает много записей, Hibernate не сможет эффективно использовать результаты из кэша. Кэш результатов запросов очищается после любой вставки, удаления или обновления в таблице, записи которой попали в кэш. Это значит, что результаты могут находиться в кэше очень недолго, и даже при повторном выполнении запроса Hibernate не будет использовать результаты кэша из-за параллельных изменений в таблицах, строки которых попали в кэш.

Для большинства запросов польза от кэширования результатов несущественна или не приносит ожидаемого эффекта. Но если ограничение в запросе использует

естественный идентификатор, как, например, `select u from User u where u.username = ?`, мы советуем использовать кэширование естественных идентификаторов, как было показано ранее в этой главе.

20.3. Резюме

- Вы узнали возможности масштабирования приложений и способы взаимодействий с большими наборами данных и параллельно работающими пользователями.
- Используя массовые операции `UPDATE` и `DELETE`, можно изменять данные прямо в базе, не теряя преимуществ JPQL и запросов на основе критериев и не используя обычного SQL.
- Вы узнали про пакетные операции с данными, позволяющие обрабатывать большие количества записей на уровне приложения.
- Мы подробно рассмотрели систему кэширования Hibernate: выборочное применение и оптимизацию общего кэша сущностей, коллекций и результатов запросов.
- Мы настроили Ehcache как механизм кэширования и узнали, как получать информацию о работе Hibernate с помощью API для сбора статистики.

Библиография

1. *Ambler S. W.* 2002. Data Modeling 101. Agile Data // www.agiledata.org/essays/dataModeling101.html.
2. *Bernard E.* 2008. Hibernate Search in Action. Manning Publications.
3. *Bloch J.* 2008. Effective Java. 2nd ed. Prentice Hall. (*Блох Д.* Java. Эффективное программирование. М.: Лори, 2016. ISBN: 978-5-85582-348-6. – *Прим. ред.*)
4. *Booch G., Rumbaugh J., Jacobson I.* 2005. The Unified Modeling Language User Guide. 2nd ed. Addison-Wesley Professional.
5. *Codd E. F.* 1970. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13 (6): 377-87 // www.acm.org/classics/nov95/toc.html.
6. *Date C. J.* 2003. An Introduction to Database Systems. 8th ed. Addison-Wesley. (*Дейт К. Дж.* Введение в системы баз данных. 8-е изд. М.: Вильямс, 2005. ISBN: 5-8459-0788-8. – *Прим. ред.*)
7. *Date C. J.* 2009. SQL and Relational Theory. O'Reilly Media. (*Дейт К. Дж.* SQL и реляционная теория. Как грамотно писать код на SQL. СПб.: Символ-Плюс, 2010. ISBN: 978-5-93286-173-8. – *Прим. ред.*)
8. *Fowler M.* 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. (*Фаулер М., Бек К., Брант Дж., Робертс Д., Апдайк У.* Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2009. ISBN: 5-93286-045-6. – *Прим. ред.*)
9. *Fowler M.* 2003. Patterns of Enterprise Application Architecture. Addison-esley Professional. (*Фаулер М.* Шаблоны корпоративных приложений. М.: Вильямс, 2009. ISBN: 978-5-8459-1611-2. – *Прим. ред.*)
10. *Gamma E., Helm R., Johnson R., Vlissides J.* 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. (*Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. М.: ДМК Пресс, 2011. ISBN: 978-5-9370-0023-1. – *Прим. ред.*)
11. *Karwin B.* 2010. SQL Antipatterns: Avoiding the Pitfalls of Database Programming. The Pragmatic Bookshelf. (*Карвин Б.* Программирование баз данных SQL. М.: Рид Групп, 2011. ISBN: 978-5-4252-0510-0. – *Прим. ред.*)
12. *Morgan T. D.* 2010. Weaning the Web off of Session Cookies: Making Digest Authentication Viable. Virtual Security Research // www.vsecurity.com/download/papers/WeaningTheWebOffOfSessionCookies.pdf.
13. *Pascal F.* 2000. Practical Issues in Database Management: A Reference for the Thinking Practitioner. Addison-Wesley Professional.
14. *Richardson L., Amundsen M., Ruby S.* 2013. RESTful Web APIs. O'Reilly Media.
15. *Shute J., et al.* 2012. F1 – The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. Research at Google // http://research.google.com/pubs/pub38125.html.
16. *Tow D.* 2003. SQL Tuning. O'Reilly Media. (*Тоу Д.* Настройка SQL: для профессионалов. СПб.: Питер, 2004. ISBN: 5-94723-959-0. – *Прим. ред.*)
17. *Walls C., Richards N.* 2004. XDoclet in Action. Manning Publications.
18. *Watterson B.* 1992. The Indispensable Calvin and Hobbes: A Calvin and Hobbes Treasury. Andrews McMeel Publishing.

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**.

Кристиан Бауэр, Гэвин Кинг, Гэри Грегори

Java Persistence API и Hibernate

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Зинкевич Д. А.*
Научный редактор *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 59,25. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф