

Java Persistence API *и Hibernate*

Кристиан Бауэр
Гэвин Кинг
Гэри Грегори



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

Original English language edition published by Manning Publications USA, USA. Copyright (c) 2015 by Manning Publications. Russian language edition copyright (c) 2015 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

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

© 2016 by Manning Publications Co.
© Оформление, издание, перевод, ДМК Пресс, 2017

Содержание

Предисловие к первому изданию	16
Введение	18
Благодарности	19
Об этой книге	21
Об изображении на обложке	24
Часть I. Начинаем работать с ORM	25
Глава 1. Основы объектно-реляционного отображения	26
1.1. Что такое долговременное хранение?	27
1.1.1. Реляционные базы данных	28
1.1.2. Разбираемся с SQL	29
1.1.3. Использование SQL в Java	30
1.2. Несоответствие парадигм	32
1.2.1. Проблема детализации	33
1.2.2. Проблема подтипов	35
1.2.3. Проблема идентичности	37
1.2.4. Проблемы, связанные с ассоциациями	38
1.2.5. Проблемы навигации по данным	39
1.3. ORM и JPA	41
1.4. Резюме	43
Глава 2. Создаем проект	44
2.1. Представляем Hibernate	44
2.2. «HELLO WORLD» и JPA	45
2.2.1. Настройка единицы хранения	46
2.2.2. Хранимый класс	47
2.2.3. Сохранение и загрузка сообщений	49
2.3. Оригинальная конфигурация Hibernate	51
2.4. Резюме	54
Глава 3. Модели предметной области и метаданные	55
3.1. Учебное приложение CaveatEmptor	56
3.1.1. Многоуровневая архитектура	56
3.1.2. Анализ предметной области	58
3.1.3. Предметная модель приложения CaveatEmptor	59

3.2. Реализация предметной модели.....	61
3.2.1. Предотвращение утечек функциональности	61
3.2.2. Прозрачность сохранения и его автоматизация	62
3.2.3. Создание классов с возможностью сохранения.....	64
3.2.4. Реализация ассоциаций в POJO.....	67
3.3. Метаданные предметной модели.....	72
3.3.1. Определение метаданных с помощью аннотаций	73
3.3.2. Применение правил валидации компонентов.....	75
3.3.3. Метаданные во внешних XML-файлах.....	78
3.3.4. Доступ к метаданным во время выполнения	82
3.4. Резюме.....	86

Часть II. Стратегии отображения 87

Глава 4. Отображение хранимых классов..... 88

4.1. Понятие сущностей и типов-значений.....	88
4.1.1. Хорошо детализированные модели предметной области.....	89
4.1.2. Определение сущностей приложения	89
4.1.3. Разделение сущностей и типов-значений.....	91
4.2. Отображение сущностей с идентичностью.....	93
4.2.1. Идентичность и равенство в Java	93
4.2.2. Первый класс сущности и его отображение	94
4.2.3. Выбор первичного ключа.....	95
4.2.4. Настройка генераторов ключей.....	97
4.2.5. Стратегии генерации идентификаторов.....	99
4.3. Способы отображений сущностей.....	103
4.3.1. Управление именами	103
4.3.2. Динамическое формирование SQL	106
4.3.3. Неизменяемые сущности	107
4.3.4. Отображение сущности в подзапрос.....	108
4.4. Резюме.....	110

Глава 5. Отображение типов-значений..... 111

5.1. Отображение полей основных типов	112
5.1.1. Переопределение настроек по умолчанию для свойств основных типов	112
5.1.2. Настройка доступа к свойствам.....	114
5.1.3. Работа с вычисляемыми полями	116
5.1.4. Преобразование значений столбцов	117
5.1.5. Значения свойств, генерируемые по умолчанию	118
5.1.6. Свойства для представления времени.....	119
5.1.7. Отображение перечислений.....	120
5.2. Отображение встраиваемых компонентов.....	121
5.2.1. Схема базы данных.....	121

5.2.2. Встраиваемые классы.....	122
5.2.3. Переопределение встроенных атрибутов.....	125
5.2.4. Отображение вложенных встраиваемых компонентов.....	126
5.3. Отображение типов Java и SQL с применением конвертеров.....	128
5.3.1. Встроенные типы.....	128
5.3.2. Создание собственных конвертеров JPA.....	135
5.3.3. Расширение Hibernate с помощью пользовательских типов.....	141
5.4. Резюме.....	148

Глава 6. Отображение наследования..... 150

6.1. Одна таблица для каждого конкретного класса и неявный полиморфизм.....	151
6.2. Одна таблица для каждого конкретного класса с объединениями.....	153
6.3. Единая таблица для целой иерархии классов.....	156
6.4. Одна таблица для каждого подкласса с использованием соединений.....	159
6.5. Смешение стратегий отображения наследования.....	163
6.6. Наследование и встраиваемые классы.....	165
6.7. Выбор стратегии.....	168
6.8. Полиморфные ассоциации.....	170
6.8.1. Полиморфная ассоциация <i>многие к одному</i> (many-to-one).....	170
6.8.2. Полиморфные коллекции.....	173
6.9. Резюме.....	174

Глава 7. Отображение коллекций и связей

между сущностями..... 175

7.1. Множества, контейнеры, списки и словари с типами-значениями.....	176
7.1.1. Схема базы данных.....	176
7.1.2. Создание и отображение поля коллекции.....	176
7.1.3. Выбор интерфейса коллекции.....	178
7.1.4. Отображение множества.....	180
7.1.5. Отображение контейнера идентификаторов.....	181
7.1.6. Отображение списка.....	182
7.1.7. Отображение словаря.....	184
7.1.8. Отсортированные и упорядоченные коллекции.....	185
7.2. Коллекции компонентов.....	188
7.2.1. Равенство экземпляров компонентов.....	189
7.2.2. Множество компонентов.....	191
7.2.3. Контейнер компонентов.....	193
7.2.4. Словарь с компонентами в качестве значений.....	194
7.2.5. Компоненты в роли ключей словаря.....	195
7.2.6. Коллекции во встраиваемых компонентах.....	197
7.3. Отображение связей между сущностями.....	198
7.3.1. Самая простая связь.....	199
7.3.2. Определение двунаправленной связи.....	200

7.3.3. Каскадная передача состояния.....	202
7.4. Резюме.....	209

Глава 8. Продвинутые приемы отображения связей

между сущностями	211
8.1. Связи <i>один к одному</i>	212
8.1.1. Общий первичный ключ	212
8.1.2. Генератор внешнего первичного ключа.....	215
8.1.3. Соединение с помощью столбца внешнего ключа	218
8.1.4. Использование таблицы соединения.....	220
8.2. Связь <i>один ко многим</i>	222
8.2.1. Применение контейнеров в связях <i>один ко многим</i>	223
8.2.2. Однонаправленное и двунаправленное отображения списка.....	224
8.2.3. Необязательная связь <i>один ко многим</i> с таблицей соединения.....	227
8.2.4. Связь <i>один ко многим</i> во встраиваемых классах.....	229
8.3. Тройные связи и связи <i>многие ко многим</i>	231
8.3.1. Однонаправленные и двунаправленные связи <i>многие ко многим</i>	232
8.3.2. Связь <i>многие ко многим</i> с промежуточной сущностью	234
8.3.3. Тройные связи с компонентами	239
8.4. Связи между сущностями с использованием словарей.....	242
8.4.1. Связь <i>один ко многим</i> со свойством для ключа.....	242
8.4.2. Тройное отношение вида ключ/значение.....	243
8.5. Резюме.....	245

Глава 9. Сложные и унаследованные схемы

9.1. Улучшаем схему базы данных.....	247
9.1.1. Добавление вспомогательных объектов базы данных	248
9.1.2. Ограничения SQL.....	251
9.1.3. Создание индексов	258
9.2. Унаследованные первичные ключи	259
9.2.1. Отображение естественных первичных ключей.....	259
9.2.2. Отображение составных первичных ключей	260
9.2.3. Внешние ключи внутри составных первичных ключей.....	262
9.2.4. Внешний ключ, ссылающийся на составной первичный ключ	266
9.2.5. Внешние ключи, ссылающиеся на непервичные ключи	267
9.3. Отображение свойств во вторичные таблицы	268
9.4. Резюме.....	270

Часть III. Транзакционная обработка данных

Глава 10. Управление данными.....

10.1. Жизненный цикл хранения.....	273
10.1.1. Состояния экземпляров сущностей	273

10.1.2. Контекст хранения.....	275
10.2. Интерфейс EntityManager.....	277
10.2.1. Каноническая форма единицы работы.....	277
10.2.2. Сохранение данных.....	279
10.2.3. Извлечение и модификация хранимых данных.....	280
10.2.4. Получение ссылки на объект.....	282
10.2.5. Переход данных во временное состояние.....	283
10.2.6. Изменение данных в памяти.....	285
10.2.7. Репликация данных.....	285
10.2.8. Кэширование в контексте хранения.....	286
10.2.9. Выталкивание контекста хранения.....	288
10.3. Работа с отсоединенным состоянием.....	289
10.3.1. Идентичность отсоединенных экземпляров.....	289
10.3.2. Реализация метода проверки равенства.....	292
10.3.3. Отсоединение экземпляров сущностей.....	295
10.3.4. Слияние экземпляров сущностей.....	296
10.4. Резюме.....	298
Глава 11. Транзакции и многопоточность.....	299
11.1. Основы транзакций.....	300
11.1.1. Атрибуты ACID.....	300
11.1.2. Транзакции в базе данных и системные транзакции.....	300
11.1.3. Программные транзакции с JTA.....	301
11.1.4. Обработка исключений.....	303
11.1.5. Декларативное определение границ транзакции.....	306
11.2. Управление параллельным доступом.....	307
11.2.1. Многопоточность на уровне базы данных.....	307
11.2.2. Оптимистическое управление параллельным доступом.....	313
11.2.3. Явные пессимистические блокировки.....	322
11.2.4. Как избежать взаимоблокировок.....	325
11.3. Доступ к данным вне транзакции.....	327
11.3.1. Чтение данных в режиме автоматического подтверждения.....	328
11.3.2. Создание очереди изменений.....	330
11.4. Резюме.....	332
Глава 12. Планы извлечения, стратегии и профили.....	333
12.1. Отложенная и немедленная загрузка.....	334
12.1.1. Прокси-объекты.....	335
12.1.2. Отложенная загрузка хранимых коллекций.....	339
12.1.3. Реализация отложенной загрузки путем перехвата вызовов.....	342
12.1.4. Немедленная загрузка коллекций и ассоциаций.....	345
12.2. Выбор стратегии извлечения.....	347
12.2.1. Проблема $n + 1$ выражений SELECT.....	347

12.2.2. Проблема декартова произведения	348
12.2.3. Массовая предварительная выборка данных	351
12.2.4. Предварительное извлечение коллекций с помощью подзапросов.....	354
12.2.5. Отложенное извлечение с несколькими выражениями SELECT.....	355
12.2.6. Динамическое немедленное извлечение	356
12.3. Профили извлечения.....	358
12.3.1. Определение профилей извлечения Hibernate.....	359
12.3.2. Графы сущностей.....	360
12.4. Резюме	364

Глава 13. Фильтрация данных.....365

13.1. Каскадная передача изменений состояния	366
13.1.1. Доступные способы каскадирования	367
13.1.2. Транзитивное отсоединение и слияние	367
13.1.3. Каскадное обновление	370
13.1.4. Каскадная репликация	372
13.1.5. Глобальное каскадное сохранение.....	373
13.2. Прием и обработка событий	374
13.2.1. Приемники событий JPA и обратные вызовы	374
13.2.2. Реализация перехватчиков Hibernate	378
13.2.3. Базовый механизм событий.....	383
13.3. Аудит и версионирование с помощью Hibernate Envers	384
13.3.1. Включение ведения журнала аудита	384
13.3.2. Ведение аудита	386
13.3.3. Поиск версий.....	387
13.3.4. Получение архивных данных	388
13.4. Динамическая фильтрация данных	391
13.4.1. Создание динамических фильтров.....	392
13.4.2. Применение фильтра.....	392
13.4.3. Активация фильтра.....	393
13.4.4. Фильтрация коллекций.....	394
13.5. Резюме	395

Часть IV. Создание запросов.....397

Глава 14. Создание и выполнение запросов.....398

14.1. Создание запросов.....	399
14.1.1. Интерфейсы запросов JPA	399
14.1.2. Результаты типизированных запросов	402
14.1.3. Интерфейсы Hibernate для работы с запросами	402
14.2. Подготовка запросов.....	404
14.2.1. Защита от атак на основе внедрения SQL-кода.....	404
14.2.2. Связывание именованных параметров.....	405

14.2.3. Связывание позиционных параметров.....	406
14.2.4. Постраничная выборка больших наборов с результатами.....	407
14.3. Выполнение запросов.....	409
14.3.1. Извлечение полного списка результатов	409
14.3.2. Получение единичных результатов	409
14.3.3. Прокрутка с помощью курсоров базы данных.....	411
14.3.4. Обход результатов с применением итератора	412
14.4. Обращение к запросам по именам и их удаление из программного кода.....	413
14.4.1. Вызов именованных запросов.....	414
14.4.2. Хранение запросов в метаданных XML.....	414
14.4.3. Хранение запросов в аннотациях.....	416
14.4.4. Программное создание именованных запросов.....	416
14.5. Подсказки для запросов	417
14.5.1. Установка предельного времени выполнения	418
14.5.2. Установка режима выталкивания контекста хранения.....	419
14.5.3. Установка режима только для чтения	419
14.5.4. Определение количества одновременно извлекаемых записей.....	420
14.5.5. Управление комментариями SQL.....	420
14.5.6. Подсказки для именованных запросов.....	421
14.6. Резюме	422
Глава 15. Языки запросов	424
15.1. Выборка.....	425
15.1.1. Назначение псевдонимов и определение корневых источников запроса	426
15.1.2. Полиморфные запросы.....	427
15.2. Ограничения.....	428
15.2.1. Выражения сравнения	430
15.2.2. Выражения с коллекциями.....	434
15.2.3. Вызовы функций.....	435
15.2.4. Упорядочение результатов запроса.....	438
15.3. Проекция	439
15.3.1. Проекция сущностей и скалярных значений	439
15.3.2. Динамическое создание экземпляров	441
15.3.3. Извлечение уникальных результатов.....	443
15.3.4. Вызов функций в проекциях.....	443
15.3.5. Агрегирующие функции	446
15.3.6. Группировка данных.....	447
15.4. Соединения	449
15.4.1. Соединения в SQL	449
15.4.2. Соединение таблиц в JPA.....	452
15.4.3. Неявные соединения по связи	452
15.4.4. Явные соединения.....	454

15.4.5. Динамическое извлечение с помощью соединений	456
15.4.6. Тета-соединения	460
15.4.7. Сравнение идентификаторов	461
15.5. Подзапросы	463
15.5.1. Коррелированные и некоррелированные подзапросы	463
15.5.2. Кванторы	464
15.6. Резюме	466
Глава 16. Дополнительные возможности запросов	467
16.1. Преобразование результатов запросов	467
16.1.1. Получение списка списков	469
16.1.2. Получение списка словарей	469
16.1.3. Отображение атрибутов в свойства компонента JavaBean	470
16.1.4. Создание преобразователя ResultTransformer	471
16.2. Фильтрация коллекций	472
16.3. Интерфейс запросов на основе критериев в Hibernate	475
16.3.1. Выборка и упорядочение	475
16.3.2. Ограничения	476
16.3.3. Проекция и агрегирование	478
16.3.4. Соединения	479
16.3.5. Подзапросы	481
16.3.6. Запросы по образцу	482
16.4. Резюме	484
Глава 17. Настройка SQL-запросов	485
17.1. Назад к JDBC	486
17.2. Отображение результатов SQL-запросов	488
17.2.1. Проекция в SQL-запросах	489
17.2.2. Отображение в классы сущностей	490
17.2.3. Настройка отображения запросов	492
17.2.4. Размещение обычных запросов в отдельных файлах	504
17.3. Настройка операций CRUD	509
17.3.1. Подключение собственных загрузчиков	509
17.3.2. Настройка операций создания, изменения, удаления	510
17.3.3. Настройка операций над коллекциями	512
17.3.4. Немедленное извлечение в собственном загрузчике	514
17.4. Вызов хранимых процедур	517
17.4.1. Возврат результата запроса	518
17.4.2. Возврат нескольких результатов и количества изменений	519
17.4.3. Передача входных и выходных аргументов	521
17.4.4. Возвращение курсора	524
17.5. Применение хранимых процедур для операций CRUD	526
17.5.1. Загрузчик, вызывающий процедуру	526

17.5.2. Использование процедур в операциях CUD	527
17.6. Резюме	529
Часть V. Создание приложений	531
Глава 18. Проектирование клиент-серверных приложений	532
18.1. Разработка уровня хранения.....	533
18.1.1. Обобщенный шаблон «объект доступа к данным»	535
18.1.2. Реализация обобщенных интерфейсов.....	537
18.1.3. Реализация интерфейсов DAO	539
18.1.4. Тестирование уровня хранения	541
18.2. Создание сервера без состояния	543
18.2.1. Редактирование информации о товаре	543
18.2.2. Размещение ставки	546
18.2.3. Анализ приложения без состояния	550
18.3. Разработка сервера с сохранением состояния	552
18.3.1. Редактирование информации о товаре	553
18.3.2. Анализ приложений с сохранением состояния	558
18.4. Резюме	561
Глава 19. Создание веб-приложений	562
19.1. Интеграция JPA и CDI	563
19.1.1. Создание экземпляра EntityManager	563
19.1.2. Присоединение экземпляра EntityManager к транзакциям.....	565
19.1.3. Внедрение экземпляра EntityManager.....	565
19.2. Сортировка и постраничная выборка данных	567
19.2.1. Реализация постраничной выборки с помощью смещения или поиска	567
19.2.2. Реализация постраничной выборки в уровне хранения.....	570
19.2.3. Постраничная выборка.....	576
19.3. Создание JSF-приложений.....	577
19.3.1. Службы с областью видимости запроса	578
19.3.2. Службы с областью видимости диалога.....	581
19.4. Сериализация данных предметной модели	590
19.4.1. Создание JAX-RS-службы.....	591
19.4.2. Применение JAXB-отображений	592
19.4.3. Сериализация прокси-объектов Hibernate.....	595
19.5. Резюме	598
Глава 20. Масштабирование Hibernate	599
20.1. Массовые и пакетные операции обработки данных	600
20.1.1. Массовые операции в запросах на основе критериев и JPQL	600
20.1.2. Массовые операции в SQL.....	605
20.1.3. Пакетная обработка данных	606

20.1.4. Интерфейс StatelessSession	610
20.2. Кэширование данных	612
20.2.1. Архитектура общего кэша в Hibernate	613
20.2.2. Настройка общего кэша	618
20.2.3. Кэширование коллекций и сущностей	619
20.2.4. Проверка работы разделяемого кэша	623
20.2.5. Установка режимов кэширования	625
20.2.6. Управление разделяемым кэшем	627
20.2.7. Кэш результатов запросов	627
20.3. Резюме	630
Библиография	631

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 продвинутое 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 Practitioner» (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 Той (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 ACID¹), 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* одновременно¹.

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 «Композовщик» (Composite), описание которых можно найти в «Design Patterns: Elements of 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.

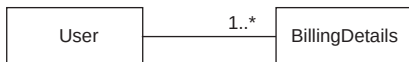


Рис. 1.1 ❖ Простая UML-диаграмма сущностей `User` и `BillingDetails`

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 отдельно 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 малоизвестной 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 следствие 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.

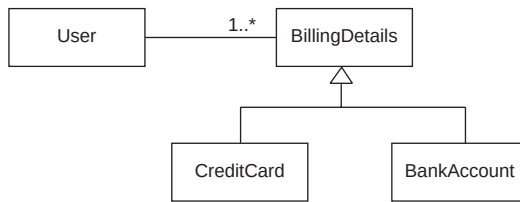


Рис. 1.3 ❖ Применение наследования для различных способов оплаты

Какие изменения требуется внести для поддержки обновленной структуры Java-классов? Требуется ли создавать таблицу `CREDITCARD`, расширяющую `BILLING-DETAILS`? 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 удастся 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.