Технологии Java для работы с базами данных

Взаимодействие с базами данных с помощью JDBC

Имеется несколько способов организовать взаимодействие с базой данных в программе на Java. Наиболее простой и прямой — использовать API JDBC (Java Database Connectivity). Более сложные, но зато более гибкие решения — использование JDO (Java Data Objects) или одной из библиотек ORM (Object-Relational Mapping), например, Hibernate.

JDBC предоставляет общий программный интерфейс (API) для доступа к базам данных в рамках различных СУБД. Для такого доступа необходимо наличие для данной СУБД драйвера, реализующего этот API.

В рамках JDBC данные представлены в той же структуре, что и в реляционных СУБД, — как записи, состоящие из нескольких полей и размещенные в таблицах. Данные могут извлекаться в виде отношений, для этого JDBC предоставляет интерфейс для выполнения запросов на SQL. В отличие от этого подхода, ORM предоставляют доступ к хранимым данным в виде объектов.

Классы и интерфейсы JDBC находятся в пакетах java.sql и javax.sql (последний входит в состав J2EE). Основной механизм получения доступа к данным — посылка запросов на SQL, оформленных в виде объектов, реализующих интерфейс Statement. Для получения таких объектов, нужно установить с базой данных соединение (представлено как объект Connection), используя классы драйвера JDBC для нужной СУБД, либо класс DriverManager, либо интерфейс DataSource. Использование последнего предпочтительно при необходимости поддерживать распределенные транзакции, пул соединений и SQL-запросов, т.е. в сложных системах, которые должны эффективно обслуживать много параллельно работающих пользователей и связываться одновременно с несколькими источниками данных.

Простейшая программа, связывающаяся с СУБД при помощи JDBC, выглядит примерно так.

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

import org.hsqldb.jdbcDriver; // используется JDBC драйвер HyperSQL

public class JDBCRequest
{
  public static void main(String[] args)
  {
    try
    {
      Properties props = new Properties();
      props.setProperty("username", "username");
      props.setProperty("password", "password");

      Connection cnx = jdbcDriver.getConnection(
                           "jdbc:hsqldb:hsql://localhost", props);
      Statement stmt = cnx.createStatement();

      // предполагаем, что есть таблица book с полями title, isbn и issuetime
      stmt.execute("SELECT title, isbn, issuetime FROM book");
      ResultSet rs = stmt.getResultSet();
```

```
      while(rs.next())
      {
        System.out.println("" + rs.getRow() + ": Title: " + rs.getString(1)
                                             + "; ISBN: " + rs.getString(2)
                                             + "; Time: " + rs.getTimestamp(3)
                           );
      }
    }
    catch (SQLException e)
    {
      e.printStackTrace();
    }
  }
}
```

Другие классы JDBC служат обертками для типов данных SQL, позволяют организовывать кэширование данных и транзакции при работе с базами данных, использовать заранее скомпилированные SQL-запросы с устанавливаемыми при их выполнении параметрами для повышения эффективности работы и пр.

Взаимодействие с базами данных с помощью Hibernate

Hibernate является примером преобразователя реляционных данных в объектное представление (Object-Relational Mapper, ORM). Для Java другими примерами такого рода являются также реализации JDO (например, JPOX), iBATIS, EclipseLink, TopLink.

ORM позволяют работать с данными как с обычными объектами, независимо от того, как они представлены в базе данных.

При этом, естественно, нужно определить классы, которые будут использоваться как типы хранимых объектов, и способ отображения таблиц и их полей на эти классы. Hibernate позволяет в качестве классов хранимых объектов использовать произвольные Java классы. Отображение полей таблиц на свойства этих классов (свойство типа Type с именем something в Java — это один или пара методов Type getSomething(), void setSomething(Type v), для свойств булевского типа вместо getSomething() пишется метод isSomething()) описывается либо в специальном формате на базе XML, либо с помощью аннотаций классов и их свойств.

Для примера рассмотрим базу данных с изображенной на Рисунке 1 схемой. Она содержит три основных таблицы, содержащие данные о книгах (book), их авторах (author) и издателях (publisher). Поскольку связь между книгами и авторами множественная с обеих сторон, введена вспомогательная таблица bookauthor.

Данные этих таблиц будем представлять в виде объектов трех классов — Book, Author и Publisher, код которых представлен ниже.

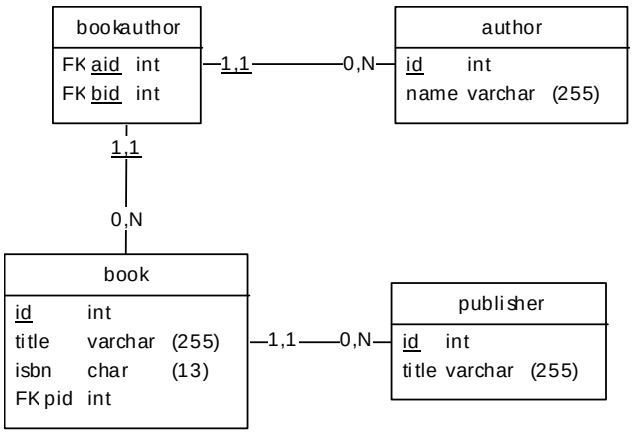

Рисунок 1. Схема базы данных

```
public class Book
{
  private int id;
  private String title;
  private String isbn;
  private Set<Author> authors;
  private Publisher publisher;

  public int getId()                { return id;    }
  public void setId(int id)         { this.id = id; }

  public String getTitle()          { return title;       }
  public void setTitle(String title) { this.title = title; }

  public String getISBN()           { return isbn;      }
  public void setISBN(String isbn)  { this.isbn = isbn; }

  public Set<Author> getAuthors()             { return authors; }
  public void setAuthors(Set<Author> authors) { this.authors = authors; }

  public Publisher getPublisher() { return publisher; }
  public void setPublisher(Publisher publisher)
  { this.publisher = publisher; }
}

public class Author
{
  private int id;
  private String name;
  private String isbn;
  private Set<Book> books;

  public int getId()              { return id;    }
  public void setId(int id)       { this.id = id; }

  public String getName()         { return name;      }
  public void setName(String name) { this.name = name; }

  public Set<Book> getBooks()           { return books; }
  public void setBooks(Set<Book> books) { this.books = books; }
}

public class Publisher
{
  private int id;
  private String title;
  private Set<Book> books;

  public int getId()                { return id;    }
  public void setId(int id)         { this.id = id; }

  public String getTitle()          { return title;       }
  public void setTitle(String title) { this.title = title; }

  public Set<Book> getBooks()           { return books; }
  public void setBooks(Set<Book> books) { this.books = books; }
}
```

Заметим, что в этом коде скалярные поля таблиц (имеющие элементарные типы, например, id, title, isbn) представлены так же, как и связи в виде свойств. Множественные связи представлены в виде коллекций объектов, связанных с данным, причем связь «многие ко многим» представлена двумя полями-коллекциями в соответствующих классах.

Чтобы определить отображение полей в свойства необходим конфигурационный файл примерно следующего вида.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="Author" table="author">
    <id name="id" column="id" type="int">
      <generator class="identity"/>
    </id>
    <property name="Name" column="name" type="string"/>
    <set name="Books" lazy="true" table="bookauthor">
      <key column="aid"/>
      <many-to-many class="Book" column="bid"/>
    </set>
  </class>
  <class name="Publisher" table="publisher">
    <id name="id" column="id" type="int">
      <generator class="identity"/>
    </id>
    <property name="Title" column="title" type="string"/>
    <set name="Books" inverse="true" lazy="true">
      <key column="pid"/>
      <one-to-many class="Book"/>
    </set>
  </class>
  <class name="Book" table="book">
    <id name="id" column="id" type="int">
      <generator class="identity"/>
    </id>
    <property name="Title" column="title" type="string"/>
    <property name="ISBN" column="isbn" type="string"/>
    <many-to-one name="Publisher" class="Publisher" column="pid"/>
    <set name="Authors" inverse="true" lazy="true" table="bookauthor">
      <key column="bid"/>
      <many-to-many class="Author" column="aid"/>
    </set>
  </class>
</hibernate-mapping>
```

В этом файле определяется соответствие между полями таблиц и свойствами классов. Если имена классов и таблиц или свойств и полей совпадают, достаточно указывать только имя класса или свойства.

При определении множественных связей используются специфические конструкции.

Чтобы получить работоспособный пример, нужен некоторый код, который будет выполнять запросы к базе данных, используя библиотеку Hibernate. Такие запросы строятся с помощью методов createQuery() и createCriteria() класса org.hibernate.Session. Пример работы с описанной базой может выглядеть так.

```
public class DBManager
{
  public static void main(String[] args)
  {
    Session s = new Configuration().configure()
                      .buildSessionFactory().getCurrentSession();

    s.beginTransaction();
    List<Book> l = s.createQuery("from Book").list();

    for(Book b : l)
```

```
    {
      System.out.println("ID: " + b.getId()
                  + "; Title: " + b.getTitle()
                  + "; ISBN: " + b.getISBN()
                  + "; Publisher: " + b.getPublisher().getTitle());
      System.out.print('{');
      for(Author a : b.getAuthors())
      {
        System.out.print("" + a.getName() + "; ");
      }
      System.out.println('}');
    }
  }
}
```

Для завершения примера необходим только еще один конфигурационный файл Hibernate, определяющий свойства соединения с базой данных. Он может выглядеть так.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
    <property name="connection.username">sa</property>

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>
    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>
    <!-- Disable the second-level cache -->
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider</property>

    <!-- Echo all executed SQL to stdout -->
    <!-- <property name="show_sql">true</property> -->

    <mapping resource="library.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Основные параметры, которые нужно установить — свойства соединения JDBC, включая класс драйвера. Обязательным параметром является диалект SQL. В данном примере использован диалект СУБД HyperSQL.

Все запросы к базе данных выполняются через объекты-сессии класса org.hibernate.Session. Для получения таких объектов используется фабрика сессий (объект типа org.hibernate.SessionFactory), которая в приложении обычно одна и получается при инициализации с помощью выполнения инструкции `new Configuration().configure().buildSessionFactory()`.

Пример запроса с параметрами может выглядеть так (получение книг данного автора).

```
public static void getAuthorBooks(String name)
{
  Session s = sfactory.getCurrentSession();
```

```
  s.beginTransaction();
  List<Book> l = s.createCriteria(Book.class)
                  .addOrder(Order.asc("Title"))
                  .createCriteria("Authors")
                  .add(Restrictions.eq("Name", name))
                  .list();

  for(Book b : l)
    System.out.println"Title: " + b.getTitle());
}
```

Ниже приведен пример метода, изменяющего данные в базе и сохраняющего изменения.

```
public static void modifyAuthor()
{
  Session s = sfactory.getCurrentSession();

  s.beginTransaction();
  Author a = (Author) s.createCriteria(Author.class)
                       .add(Restrictions.eq("Name", "Andrew Tanenbaum"))
                       .uniqueResult();

  System.out.println("ID: " + a.getId() + "; Name: " + a.getName());

  a.setName("A. Tanenbaum");
  s.saveOrUpdate(a);

  s.getTransaction().commit();
}
```

Метод saveOrUpdate() объекта-сессии позволяет сохранить изменения и новые объекты. Отдельные методы save() и update() сохраняют новые объекты (записи, соответствующие им) и вносят изменения в уже имеющиеся данные. Метод delete() позволяет удалить объект. Метод persist() делает временно созданный объект (не связанный ни с какой записью) хранимым.

Все действия по модификации данных должны выполняться внутри транзакций, которые в случае успешного выполнения должны завершаться вызовом метода commit(). При возникновении непредвиденных ситуаций, обычно проявляющихся в виде исключений, сделанные изменения нужно отменять при помощи rollback().

Литература

[1] Документация по JDBC на сайте SUN Developer Network. http://java.sun.com/products/jdbc/download.html.

[2] Руководство по JDBCв рамках Java Tutorial. http://java.sun.com/docs/books/tutorial/jdbc/.

[3] Б. Ван Хейк. JDBC. Java и базы данных. Лори, 2000.

[4] Сайт Hibernate. https://www.hibernate.org/.

[5] Справочная документация по Hibernate. http://docs.jboss.org/hibernate/stable/core/reference/en/html/.

[6] C. Bauer, G. King. Java Persistence with Hibernate. Manning, 2006.

[7] D. Minter, J. Linwood. Pro Hibernate 3. Apress, 2005.

[8] А. Хемраджани. Гибкая разработка приложений на Java с помощью Spring, Hibernate и Eclipse. Вильямс, 2008.