目录

第一篇：先导篇：为什么有数据库访问技术？他给我们带来了什么好处?

本篇介绍数据库访问技术的背景、发展及其对开发效率和软件架构的影响。

第一章：ORM概述

- 介绍ORM（对象关系映射）的概念和原理。
- 为什么需要ORM？它如何简化数据库操作？
- 传统SQL操作 vs ORM方式。

第二章：数据库访问技术速览与比较

- 概览JDBC、Hibernate、MyBatis、Spring Data JPA等主流数据访问技术。
- 介绍各个框架的特点、适用场景和优缺点。

第二篇：实战篇：深入理解Java世界中的主流数据访问框架

本篇专注于实践，深入讲解JDBC和三大主流ORM框架的实现方式和用法。

第三章：JDBC 主要对象

- 讲解JDBC的核心组件，如Connection、Statement、ResultSet等。
- 介绍如何使用JDBC进行数据库操作。

第四章：深入理解Hibernate

- 详细介绍Hibernate的配置、实体映射、查询语言（HQL）、事务管理等。
- 讨论Hibernate的缓存机制和优化策略。

第五章：深入理解MyBatis

- 介绍MyBatis的XML配置、注解方式、动态SQL等。
- 讲解MyBatis与Spring的集成。

第六章：你应该知道的JPA

- 讲解JPA的核心概念和规范。

- 介绍JPA的查询语言（JPQL）、实体生命周期和关系映射。

第三篇：进阶篇：与框架设计结合、深入实现原理

本篇从架构设计的角度，深入探讨数据库访问技术的高级主题和实现原理。

第七章：Repository模式理论&实现

- 介绍Repository模式的设计思想。
- 讲解如何在Spring Data JPA和其他框架中实现Repository模式。
- 造轮子：使用MyBatis实现Repository模式

第八章：事务处理&Unit Of Work

- 介绍事务的基本概念（ACID）。
- 讲解Unit Of Work模式及其在Spring/Hibernate中的应用。

第九章：非关系型数据处理

- 介绍NoSQL数据库（MongoDB、Redis等）的数据访问方式。
- 讲解如何在Spring和Java环境中操作NoSQL数据。

说明：前两篇是初级人员必学内容。

1. ORM的概念和基本工作原理

对象关系映射（Object-Relational Mapping，简称 **ORM**）是一种技术，它通过程序代码中的对象与数据库中的表进行映射，使开发者能够以面向对象的方式操作数据库。

ORM框架负责将对象转换为SQL语句，并将查询结果映射回对象，从而简化数据库操作。
在关系型数据库中，数据以表的形式存储，而在面向对象编程中，数据是以对象的形式存在的。ORM 的核心工作原理是：

- 类（Class） 对应 数据表（Table）
- 对象（Object） 对应 表中的行（Row）
- 对象的属性（Fields） 对应 表的列（Columns）

例如，假设有一个 User 类，它的属性 id、name、age 分别对应数据库表 users 的列 id、name、age。ORM 框架会自动管理这些映射，使得开发者可以像操作 Java 对象一样操作数据库。

2. 为什么需要ORM？它如何简化数据库操作？

在没有 ORM 的情况下，开发者通常需要使用 **JDBC** 直接执行 SQL 语句进行数据库操作，这会带来一些问题：

- 开发效率低：手写 SQL 需要手动处理数据库连接、SQL 语法、结果集转换等，代码冗长且容易出错。
- SQL注入风险：手动拼接 SQL 字符串容易导致安全漏洞，ORM 通过参数绑定方式防止 SQL 注入。
- 数据库依赖性强：手写 SQL 可能会绑定到某个特定数据库，而 ORM 框架可以适配不同的数据库，使代码更具可移植性。
- 维护成本高：如果数据库结构发生变化，使用 ORM 可以更容易地适应变化，而手写 SQL 可能需要大规模修改代码。

ORM 通过提供 **自动SQL生成**、**事务管理**、**对象映射** 等功能，使得开发者可以直接操作对象，而 ORM 框架会在后台自动转换成 SQL 语句，提高开发效率和可维护性。

3. 传统SQL操作 vs ORM方式

传统 SQL 操作

使用 JDBC 执行 SQL 需要编写大量的代码，手动处理数据库连接、SQL 语句和结果集。例如：

```java
import java.sql.*;

public class JDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/testdb";
        String user = "root";
        String password = "password";

        try (Connection conn = DriverManager.getConnection(url, user, password);
             PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users WHERE age > ?")) {

            stmt.setInt(1, 25);
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                int age = rs.getInt("age");
                System.out.println("用户ID: " + id + ", 姓名: " + name + ", 年龄: " + age);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

问题:

1. 需要手动管理数据库连接（Connection）、语句（Statement）和结果集（ResultSet）。
2. 手动拼接和执行 SQL 语句，代码量大且容易出错。
3. 代码可读性较低，维护成本高。

使用 ORM 方式

使用 Hibernate，开发者可以像操作 Java 对象一样进行数据库查询，而不需要直接写 SQL。例如:

定义实体类（映射到数据库表）


```java
import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "name")
    private String name;

    @Column(name = "age")
    private int age;

    // Getters and Setters
}

```

使用 Hibernate 进行查询

```java
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

import java.util.List;

public class HibernateExample {
    public static void main(String[] args) {
        SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(User.class).buildSessionFactory();
        Session session = factory.getCurrentSession();

        try {
            session.beginTransaction();
            List<User> users = session.createQuery("FROM User u WHERE u.age > 25", User.class).getResultList();
            for (User user : users) {
                System.out.println("用户ID: " + user.getId() + ", 姓名: " + user.getName() + ", 年龄: " + user.getAge());
            }
            session.getTransaction().commit();
        } finally {
            factory.close();
        }
    }
}

```

对比:

- JDBC：需要手动编写 SQL 并管理连接，代码量大且容易出错。
- Hibernate ORM：使用面向对象方式查询数据，代码更简洁，SQL 由框架自动生成，提高了可维护性。

---

4. ORM的优势与缺点

优势

1. 开发效率高：减少样板代码，使开发者更专注于业务逻辑。

2. 数据库无关性：ORM 使得同一套代码可以适配不同的数据库，而无需修改 SQL 语句。

3. 安全性更高：自动处理 SQL 注入风险，提高系统安全性。
4. 自动管理事务：框架提供了内置的事务支持，使得事务管理更加简单。
5. 良好的扩展性：支持复杂的对象映射、级联操作和缓存等高级功能。

缺点

1. 学习成本：ORM 框架（如 Hibernate）需要学习新的 API 和配置方式。
2. 性能开销：相比于手写 SQL，ORM 可能会生成非最优的 SQL，影响性能。
3. 对复杂查询支持有限：ORM 适合简单 CRUD 操作，但在处理复杂查询时可能会遇到局限，需要使用原生 SQL。

---

5. 结论

ORM 通过对象与关系数据库之间的映射，使得开发者可以更方便地操作数据库，极大提高了开发效率和可维护性。在大多数业务应用中，ORM 是更优的选择。然而，对于需要极致优化的性能场景，可能仍然需要结合原生 SQL 进行优化。因此，合理选择和使用 ORM 是关键。

- 概览JDBC、Hibernate、MyBatis、Spring Data JPA等主流数据访问技术。
- 介绍各个框架的特点、适用场景和优缺点。

在 Java 生态中，数据库访问技术是应用程序与数据库交互的核心。不同的技术适用于不同的场景，开发者需要根据项目需求选择合适的技术。本章将介绍 JDBC、Hibernate、MyBatis、Spring Data JPA 这四种主流数据库访问技术，并通过对比表格帮助读者快速理解它们的优缺点和适用场景。

---

1. JDBC（Java Database Connectivity）

概述：
JDBC 是 Java 提供的最基础的数据库访问技术，它通过标准的 API 接口与数据库进行交互。JDBC 需要开发者手动编写 SQL 语句并处理数据库连接、结果集等底层操作。

适用场景：

- 需要直接控制 SQL 语句的场景。
- 对性能要求极高，且需要精细优化的场景。
- 学习数据库访问技术的入门工具。

优点：

- 直接操作 SQL，灵活性高。
- 不依赖任何框架，轻量级。
- 适合理解数据库访问的底层原理。

缺点：

- 代码冗余，需要手动管理连接、事务等。
- 容易出错，比如资源未关闭、SQL 注入等问题。
- 开发效率低，不适合快速迭代的项目。

---

2. Hibernate

概述：
Hibernate 是一个全自动的 ORM（对象关系映射）框架，它将 Java 对象与数据库表进行映射，开发者可以通过操作对象来实现数据库的增删改查，而无需直接编写 SQL。

适用场景：

- 需要快速开发的复杂业务系统。
- 数据库表结构相对稳定，且对象与表结构高度匹配的场景。
- 需要跨数据库支持的项目。

优点：

- 自动化程度高，减少大量重复代码。

- 支持缓存机制，提升性能。
- 提供 HQL（Hibernate Query Language），简化复杂查询。
- 支持事务管理和延迟加载。

缺点：

- 学习曲线较陡，配置复杂。
- 对复杂 SQL 的支持较弱，性能调优需要一定经验。
- 自动生成的 SQL 可能不够高效。

---

3. MyBatis

概述：
MyBatis 是一个半自动的 ORM 框架，它将 SQL 语句与 Java 对象进行映射，开发者需要编写 SQL，但 MyBatis 会自动处理结果集与对象的映射。

适用场景：

- 需要精细控制 SQL 语句的场景。
- 对 SQL 性能要求较高的项目。
- 数据库表结构与对象模型差异较大的场景。

优点：

- 灵活性高，支持复杂 SQL 和动态 SQL。
- 配置简单，易于上手。
- 性能较好，适合对 SQL 有优化需求的场景。

缺点：

- 需要手动编写 SQL，开发效率略低。
- 对于简单 CRUD 操作，代码量较多。
- 缓存机制不如 Hibernate 强大。

---

4. Spring Data JPA

概述：
Spring Data JPA 是 Spring 生态中的 ORM 框架，基于 JPA（Java Persistence API）规范，进一步简化了数据库操作。它通过 Repository 接口自动生成 SQL，开发者只需定义接口即可完成数据库操作。

适用场景：

- 需要快速开发的简单 CRUD 操作。
- 基于 Spring 生态的项目。
- 对数据库操作抽象化要求较高的场景。

优点：

- 开发效率极高，减少大量模板代码。

- 与 Spring 生态无缝集成。
- 支持方法名自动生成查询，简化查询操作。
- 提供分页、排序等常用功能。

缺点：

- 对复杂 SQL 的支持较弱。
- 学习曲线较高，尤其是对 JPA 规范的理解。
- 性能调优需要一定经验。

对比表格

技术	JDBC	Hibernate	MyBatis	Spring Data JPA
类型	手动 SQL 操作	全自动 ORM	半自动 ORM	基于 JPA 的 ORM
灵活性	极高	较低	高	较低
开发效率	低	高	中	高
性能	高（需手动优化）	中（自动生成 SQL 可能不高效）	高（可手动优化 SQL）	中（自动生成 SQL 可能不高效）
学习曲线	低	高	中	高
适用场景	需要精细控制 SQL 的场景	复杂业务系统	需要精细控制 SQL 的场景	简单 CRUD 操作
缓存支持	无	强大	较弱	中等
事务管理	手动管理	自动管理	手动或集成 Spring 管理	自动管理
跨数据库支持	需要手动适配	支持	支持	支持

总结

- JDBC：适合需要直接控制 SQL 的场景，但开发效率低。
- Hibernate：适合复杂业务系统，开发效率高，但学习曲线陡。
- MyBatis：适合需要精细控制 SQL 的场景，灵活性高。
- Spring Data JPA：适合快速开发的简单 CRUD 操作，与 Spring 生态无缝集成。

开发者应根据项目需求、团队技术栈和性能要求选择合适的技术。对于初学者，建议从 JDBC 入手，逐步学习 ORM 框架；对于快速开发项目，Spring Data JPA 和 Hibernate 是不错的选择；而对于需要精细控制 SQL 的场景，MyBatis 是更好的选择。

JDBC（Java Database Connectivity）是 Java 提供的一套用于与数据库进行交互的 API。它是 Java 数据库访问的基础，开发者可以通过 JDBC 直接操作数据库。本章将介绍 JDBC 的核心组件、基本用法以及一些常见的优化技巧。

---

1. JDBC 核心组件

JDBC 的核心组件包括以下几个类或接口：

1. DriverManager：用于管理数据库驱动，负责建立与数据库的连接。
   - 常用方法：getConnection(url, user, password)。
2. Connection：表示与数据库的连接，用于创建 Statement 对象并管理事务。
   - 常用方法：createStatement()、prepareStatement(sql)、setAutoCommit(boolean)、commit()、rollback()。
3. Statement：用于执行静态 SQL 语句并返回结果。
   - 常用方法：executeQuery(sql)、executeUpdate(sql)、execute(sql)。
4. PreparedStatement：是 Statement 的子接口，用于执行预编译的 SQL 语句，支持参数化查询，防止 SQL 注入。
   - 常用方法：setXxx(int parameterIndex, Xxx value)、executeQuery()、executeUpdate()。
5. ResultSet：表示 SQL 查询的结果集，提供了遍历和获取数据的方法。
   - 常用方法：next()、getXxx(int columnIndex)、getXxx(String columnName)。
6. SQLException：JDBC 操作中可能抛出的异常，用于处理数据库访问中的错误。

---

2. JDBC 基础操作示例

以下是一个简单的 JDBC 操作示例，展示了如何连接数据库、执行查询和更新操作。


```
import java.sql.*;

public class JDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String user = "root";
        String password = "password";

        try (Connection connection = DriverManager.getConnection(url, user, password))
{
            // 1. 创建 Statement 对象
            Statement statement = connection.createStatement();

            // 2. 执行查询
            String query = "SELECT id, name FROM users";
            ResultSet resultSet = statement.executeQuery(query);

            // 3. 遍历结果集
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                System.out.println("ID: " + id + ", Name: " + name);
            }

            // 4. 执行更新
            String update = "UPDATE users SET name = 'John Doe' WHERE id = 1";
            int rowsAffected = statement.executeUpdate(update);
            System.out.println("Rows affected: " + rowsAffected);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

---

3. JDBC 优化技巧

3.1 使用连接池

频繁创建和关闭数据库连接会消耗大量资源，使用连接池可以复用连接，提升性能。常见的连接池有：

- HikariCP：高性能的连接池。
- Apache DBCP：Apache 提供的连接池。
- C3P0：老牌连接池，功能丰富。

以下是使用 HikariCP 的示例：

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;

public class HikariExample {
    public static void main(String[] args) {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/mydatabase");
        config.setUsername("root");
        config.setPassword("password");

        try (HikariDataSource dataSource = new HikariDataSource(config);
             Connection connection = dataSource.getConnection()) {
            System.out.println("Connection obtained from pool!");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

3.2 使用 PreparedStatement

PreparedStatement 可以预编译 SQL 语句，提升性能并防止 SQL 注入。

```
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {
    preparedStatement.setString(1, "Alice");
    preparedStatement.setString(2, "alice@example.com");
    preparedStatement.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
```

3.3 批量处理

对于大量数据操作，使用批量处理可以显著提升性能。

```
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {
    for (int i = 1; i <= 1000; i++) {
        preparedStatement.setString(1, "User" + i);
        preparedStatement.setString(2, "user" + i + "@example.com");
        preparedStatement.addBatch(); // 添加到批处理
    }
    preparedStatement.executeBatch(); // 执行批处理
} catch (SQLException e)
    { e.printStackTrace();
}
```

3.4 事务管理

通过 Connection 对象管理事务，确保数据一致性。


```
try {
    connection.setAutoCommit(false); // 关闭自动提交

    // 执行多个操作
    statement.executeUpdate("UPDATE account SET balance = balance - 100 WHERE id = 1");
    statement.executeUpdate("UPDATE account SET balance = balance + 100 WHERE id = 2");

    connection.commit(); // 提交事务
} catch (SQLException e) {
    connection.rollback(); // 回滚事务
    e.printStackTrace();
} finally {
    connection.setAutoCommit(true); // 恢复自动提交
}
```

---

4. 总结

- JDBC 是 Java 数据库访问的基础，核心组件包括 DriverManager、Connection、Statement、PreparedStatement 和 ResultSet。
- 通过连接池、PreparedStatement、批量处理和事务管理，可以显著提升 JDBC 的性能和可靠性。
- 虽然 JDBC 提供了最大的灵活性，但在实际开发中，通常会结合 ORM 框架（如 Hibernate、MyBatis）来简化操作。

Hibernate 是一个全自动的 ORM（对象关系映射）框架，它将 Java 对象与数据库表进行映射，开发者可以通过操作对象来实现数据库的增删改查，而无需直接编写 SQL。本章将深入探讨 Hibernate 的核心特性，包括配置、实体映射、HQL、事务管理、缓存机制和优化策略。

---

1. Hibernate 配置

Hibernate 的配置可以通过 XML 文件或注解方式完成。以下是两种配置方式的示例：

1.1 XML 配置（hibernate.cfg.xml）

```xml
<hibernate-configuration>
    <session-factory>
        <!-- 数据库连接配置 -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydatabase</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">password</property>

        <!-- 方言配置 -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>

        <!-- 其他配置 -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

        <!-- 实体类映射 -->
        <mapping class="com.example.User"/>
    </session-factory>
</hibernate-configuration>
```

1.2 注解配置（基于 Java 配置）


```java
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new Configuration()
                    .configure("hibernate.cfg.xml") // 加载配置文件
                    .addAnnotatedClass(User.class)  // 添加实体类
                    .buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

---

2. 实体映射

Hibernate 通过注解或 XML 文件将 Java 对象映射到数据库表。以下是使用注解的示例：

```java
import javax.persistence.*;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

    // Getters and Setters
}
```

- @Entity：标记类为实体类。
- @Table：指定映射的数据库表。
- @Id：标记主键字段。
- @GeneratedValue：指定主键生成策略。
- @Column：指定字段与数据库列的映射。

---

3. 查询语言（HQL）

HQL（Hibernate Query Language）是 Hibernate 提供的面向对象的查询语言，类似于 SQL，但操作的是对象而非表。

3.1 基本查询

```java
String hql = "FROM User WHERE name = :name";
Query<User> query = session.createQuery(hql, User.class);
query.setParameter("name", "Alice");
List<User> users = query.getResultList();
```

3.2 分页查询


```java
String hql = "FROM User";
Query<User> query = session.createQuery(hql, User.class);
query.setFirstResult(0); // 起始位置
query.setMaxResults(10); // 每页大小
List<User> users = query.getResultList();
```

3.3 聚合查询

```java
String hql = "SELECT COUNT(*) FROM User";
Query<Long> query = session.createQuery(hql, Long.class);
Long count = query.getSingleResult();
```

---

4. 事务管理

Hibernate 的事务管理可以通过 Session 对象实现。以下是一个简单的事务管理示例：

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction transaction = null;

try {
    transaction = session.beginTransaction();

    // 执行操作
    User user = new User();
    user.setName("Bob");
    user.setEmail("bob@example.com");
    session.save(user);

    transaction.commit(); // 提交事务
} catch (Exception e) {
    if (transaction != null) {
        transaction.rollback(); // 回滚事务
    }
    e.printStackTrace();
} finally {
    session.close(); // 关闭 Session
}
```

---

5. Hibernate 缓存机制

Hibernate 提供了两级缓存机制，用于提升性能：

5.1 一级缓存（Session 缓存）

- 默认开启，生命周期与 Session 相同。
- 在同一个 Session 中，相同的查询只会执行一次。

5.2 二级缓存（SessionFactory 缓存）

- 需要显式配置，生命周期与 SessionFactory 相同。
- 支持第三方缓存实现，如 EhCache、Infinispan。

配置二级缓存（以 EhCache 为例）：

1. 添加依赖：


```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.6.0.Final</version>
</dependency>
```

2. 在 hibernate.cfg.xml 中启用二级缓存：

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFa
ctory</property>
```

3. 在实体类上启用缓存：

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class User { ... }
```

---

6. Hibernate 优化策略

6.1 延迟加载（Lazy Loading）

- 默认情况下，Hibernate 使用延迟加载关联对象，避免不必要的查询。
- 可以通过 @ManyToOne(fetch = FetchType.LAZY) 或 @OneToMany(fetch = FetchType.LAZY) 配置。

6.2 批量处理

- 使用 Session 的 flush() 和 clear() 方法批量处理数据，减少内存占用。

```java
for (int i = 0; i < 1000; i++) {
    User user = new User();
    user.setName("User" + i);
    session.save(user);

    if (i % 50 == 0) { // 每 50 条数据刷新一次
        session.flush();
        session.clear();
    }
}
```

6.3 使用原生 SQL

- 对于复杂查询，可以使用原生 SQL 提升性能。

```java
String sql = "SELECT * FROM users WHERE name = :name";
NativeQuery<User> query = session.createNativeQuery(sql, User.class);
query.setParameter("name", "Alice");
List<User> users = query.getResultList();
```

---

7. 总结

- Hibernate 通过 ORM 技术简化了数据库操作，开发者可以通过操作对象来实现数据库的增删改查。
- HQL 提供了面向对象的查询语言，支持复杂查询和分页。
- 事务管理和缓存机制是 Hibernate 的核心特性，能够提升性能和保证数据一致性。
- 通过延迟加载、批量处理和使用原生 SQL，可以进一步优化 Hibernate 的性能。

第五章：深入理解 MyBatis

MyBatis 是一个半自动的 ORM 框架，它将 SQL 语句与 Java 对象进行映射，开发者需要编写 SQL，但 MyBatis 会自动处理结果集与对象的映射。MyBatis 以其灵活性和对 SQL 的精细控制而著称，适合需要直接操作 SQL 的场景。本章将深入探讨 MyBatis 的核心特性，包括 XML 配置、注解方式、动态 SQL，以及如何与 Spring 集成。

---

1. MyBatis 的 XML 配置

MyBatis 的核心配置文件是 mybatis-config.xml，用于配置数据源、事务管理、映射文件等。

1.1 基本配置示例

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 环境配置 -->
    <environments default="development">
        <environment id="development">
            <!-- 事务管理器 -->
            <transactionManager type="JDBC"/>
            <!-- 数据源 -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mydatabase"/>
                <property name="username" value="root"/>
                <property name="password" value="password"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 映射文件配置 -->
    <mappers>
        <mapper resource="com/example/UserMapper.xml"/>
    </mappers>
</configuration>
```

1.2 映射文件配置

映射文件（如 UserMapper.xml）用于定义 SQL 语句和结果映射。

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.UserMapper">
    <!-- 查询用户 -->
    <select id="selectUser" resultType="com.example.User">
        SELECT * FROM users WHERE id = #{id}
    </select>

    <!-- 插入用户 -->
    <insert id="insertUser" parameterType="com.example.User">
        INSERT INTO users (name, email) VALUES (#{name}, #{email})
    </insert>

    <!-- 更新用户 -->
    <update id="updateUser" parameterType="com.example.User">
        UPDATE users SET name = #{name}, email = #{email} WHERE id = #{id}
    </update>

    <!-- 删除用户 -->
    <delete id="deleteUser" parameterType="int">
        DELETE FROM users WHERE id = #{id}
    </delete>
</mapper>
```

---

2. MyBatis 的注解方式

MyBatis 支持通过注解方式定义 SQL 语句，无需编写 XML 映射文件。

2.1 基本注解示例


```java
import org.apache.ibatis.annotations.*;

public interface UserMapper {
    @Select("SELECT * FROM users WHERE id = #{id}")
    User selectUser(int id);

    @Insert("INSERT INTO users (name, email) VALUES (#{name}, #{email})")
    void insertUser(User user);

    @Update("UPDATE users SET name = #{name}, email = #{email} WHERE id = #{id}")
    void updateUser(User user);

    @Delete("DELETE FROM users WHERE id = #{id}")
    void deleteUser(int id);
}
```

2.2 注解与 XML 的对比

- 注解方式：适合简单的 SQL 语句，代码简洁，但复杂 SQL 不易维护。
- XML 方式：适合复杂的 SQL 语句，易于维护，但需要额外的 XML 文件。

---

3. MyBatis 的动态 SQL

MyBatis 提供了强大的动态 SQL 功能，可以根据条件动态生成 SQL 语句。

3.1 常用动态 SQL 标签

- <if>：根据条件判断是否包含某段 SQL。
- <choose>、<when>、<otherwise>：类似于 Java 的 switch-case 语句。
- <where>：自动处理 WHERE 子句的前缀。
- <set>：自动处理 UPDATE 语句中的逗号
- <foreach>：遍历集合，生成 IN 子句。

3.2 动态 SQL 示例

```xml
<select id="selectUsers" resultType="com.example.User">
    SELECT * FROM users
    <where>
        <if test="name != null">
            AND name = #{name}
        </if>
        <if test="email != null">
            AND email = #{email}
        </if>
    </where>
</select>

<update id="updateUser" parameterType="com.example.User">
    UPDATE users
    <set>
        <if test="name != null">name = #{name},</if>
        <if test="email != null">email = #{email},</if>
    </set>
    WHERE id = #{id}
</update>

<select id="selectUsersByIds" resultType="com.example.User">
    SELECT * FROM users
    WHERE id IN
    <foreach collection="ids" item="id" open="(" separator="," close=")">
        #{id}
    </foreach>
</select>
```

---

4. MyBatis 与 Spring 的集成

MyBatis 可以与 Spring 无缝集成，简化配置和管理。

4.1 添加依赖

在 pom.xml 中添加 MyBatis 和 Spring 的依赖：


```xml
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.7</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>2.0.6</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.10</version>
</dependency>
```

4.2 Spring 配置

在 Spring 配置文件中配置 MyBatis 的数据源和 SqlSessionFactory。

```xml
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydatabase"/>
    <property name="username" value="root"/>
    <property name="password" value="password"/>
</bean>

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mapperLocations" value="classpath*:com/example/*Mapper.xml"/>
</bean>

<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="com.example.UserMapper"/>
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
```

```
</bean>
```


4.3 在 Spring 中使用 MyBatis

通过 Spring 注入 Mapper 接口，直接调用数据库操作。

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private UserMapper userMapper;

    public User getUserById(int id) {
        return userMapper.selectUser(id);
    }

    public void addUser(User user) {
        userMapper.insertUser(user);
    }
}
```

---

5. 总结

- MyBatis 通过 XML 或注解方式定义 SQL 语句，提供了灵活的数据库操作方式。
- 动态 SQL 功能可以根据条件动态生成 SQL 语句，适合复杂查询。
- MyBatis 与 Spring 的集成简化了配置和管理，适合在 Spring 项目中使用。

第六章：你应该知道的 JPA

JPA（Java Persistence API）是 Java EE（现为 Jakarta EE）提供的一套 ORM（对象关系映射）规范，旨在简化数据库操作。Spring Data JPA 是基于 JPA 规范的进一步封装，提供了更高级的抽象和便利功能。本章将分为三部分：首先介绍 JPA 规范，然后概述常见的 JPA 实现框架，最后重点介绍 Spring Data JPA 及其特色。

---

1. JPA 规范简介

JPA 是一套标准的 ORM 规范，定义了如何将 Java 对象映射到关系型数据库中的表，并提供了一套统一的 API 来操作数据库。以下是 JPA 规范的核心内容：

1.1 实体类（Entity）

实体类是映射到数据库表的 Java 类，使用 @Entity 注解标记。

```
import javax.persistence.*;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

    // Getters and Setters
}
```

- @Entity：标记类为实体类。
- @Table：指定映射的数据库表。
- @Id：标记主键字段。
- @GeneratedValue：指定主键生成策略。
- @Column：指定字段与数据库列的映射。

1.2 实体管理器（EntityManager）

EntityManager 是 JPA 的核心接口，用于管理实体的生命周期和执行数据库操作。

```
import javax.persistence.*;

public class UserService {
    @PersistenceContext
    private EntityManager entityManager;

    public User findUserById(Long id) {
        return entityManager.find(User.class, id);
    }

    public void saveUser(User user) {
        entityManager.persist(user);
    }
}
```

- persist()：将实体对象持久化到数据库。
- find()：根据主键查询实体对象。
- merge()：更新实体对象。
- remove()：删除实体对象。

1.3 JPQL（Java Persistence Query Language）

JPQL 是 JPA 提供的面向对象的查询语言，类似于 SQL，但操作的是实体类和属性。

```
String jpql = "SELECT u FROM User u WHERE u.name = :name";
TypedQuery<User> query = entityManager.createQuery(jpql, User.class);
query.setParameter("name", "Alice");
List<User> users = query.getResultList();
```

1.4 实体生命周期

JPA 实体的生命周期包括以下状态：

- 新建（New）：对象刚创建，未与 EntityManager 关联。
- 托管（Managed）：对象被 EntityManager 管理，处于持久化上下文中。
- 游离（Detached）：对象与 EntityManager 断开连接，但仍存在于数据库中。
- 删除（Removed）：对象被标记为删除，将在事务提交时从数据库中删除。

1.5 关系映射

JPA 支持实体类之间的多种关系映射，包括一对一、一对多、多对一和多对多。

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany(mappedBy = "user")
    private List<Order> orders;
}

@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
}
```


2. JPA 实现框架概览

JPA 规范需要具体的实现框架来提供功能。以下是一些常见的 JPA 实现框架：

2.1 Hibernate

- 最流行的 JPA 实现框架。
- 提供了丰富的功能，如缓存、延迟加载、批量处理等。
- 支持 JPA 规范，同时也提供了自己的扩展功能（如 HQL、Criteria API）。

2.2 EclipseLink

- Eclipse 基金会开发的 JPA 实现。
- 功能强大，支持 JPA 规范以及 EclipseLink 的扩展功能。
- 常用于 Java EE 应用服务器（如 GlassFish）。

2.3 OpenJPA

- Apache 开发的 JPA 实现。
- 提供了对 JPA 规范的支持，适合需要高度定制化的场景。

2.4 DataNucleus

- 支持 JPA 规范，同时也支持 JDO（Java Data Objects）规范。

- 适用于需要同时支持关系型数据库和非关系型数据库的场景。

---

3. Spring Data JPA 及其特色

Spring Data JPA 是基于 JPA 规范的进一步封装，旨在简化数据访问层的开发。以下是 Spring Data JPA 的核心特色：

3.1 Repository 接口

Spring Data JPA 提供了 Repository 接口，开发者只需定义接口即可完成数据库操作。

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByName(String name);
}
```

- JpaRepository：提供了常用的 CRUD 操作方法。
- 方法名自动生成查询：如 findByName(String name)。

3.2 方法名自动生成查询

Spring Data JPA 支持根据方法名自动生成查询，无需编写 SQL 或 JPQL。

```
List<User> findByNameAndEmail(String name, String email);
```

3.3 分页和排序

Spring Data JPA 提供了对分页和排序的内置支持。

```
Page<User> findAll(Pageable pageable);
List<User> findAll(Sort sort);
```

3.4 与 Spring 的集成

Spring Data JPA 与 Spring 生态无缝集成，支持依赖注入和事务管理。

```
import org.springframework.beans.factory.annotation.Autowired;
im ort org.springframework.stereotype.Service;

@S rvice
pu lic class UserService {
    @Autowired
    private UserRepository userRepository;

    public List<User> findUsersByName(String name) {
        return userRepository.findByName(name);
    }
}
```

3.5 自定义查询

Spring Data JPA 支持通过 @Query 注解定义自定义查询。

```
@Query("SELECT u FROM User u WHERE u.email LIKE %:email%")
List<User> findByEmailContaining(@Param("email") String email);
```

---

4. 总结

- JPA 规范：定义了标准的 ORM 接口和注解，旨在简化数据库操作。
- JPA 实现框架：如 Hibernate、EclipseLink 等，提供了对 JPA 规范的具体实现。
- Spring Data JPA：基于 JPA 规范的进一步封装，提供了 Repository 接口、方法名查询、分页排序等高级功能，极大地简化了数据访问层的开发。

第七章：Repository模式理论&实现

7.1 Repository模式概述

7.1.1 什么是Repository模式?

Repository模式是一种面向领域驱动设计（DDD）的数据访问模式，它在应用程序与数据源（数据库、API等）之间提供了一个抽象层，封装了数据的存取逻辑。

7.1.2 Repository模式的核心思想

- 将数据访问逻辑封装到专门的Repository类中，而非直接在业务逻辑代码中操作数据库。
- 通过Repository提供统一的接口，使业务逻辑无需关心底层数据库访问方式。
- 促进单一职责原则（SRP），增强代码的可维护性和可测试性。

7.2 Repository模式的优势

- 降低耦合

：业务逻辑代码无需直接操作数据库，只需与Repository交互。

- 提高可测试性

：可以轻松地使用Mock对象进行单元测试。

- 增强可维护性

：数据库操作代码集中在Repository中，方便维护和替换。

- 支持不同的持久化技术

：可适配JPA、Hibernate、MyBatis、JDBC等多种数据访问技术。

7.3 Repository模式的接口和设计逻辑

7.3.1 定义通用的Repository接口

```
public interface Repository<T, ID> {
    T findById(ID id);
    List<T> findAll();
    void save(T entity);
    void update(T entity);
    void delete(ID id);
}
```

该接口定义了基本的数据访问操作，所有具体的Repository实现都可以继承这个接口。

7.3.2 创建具体的实体Repository接口

假设我们有一个User实体类，我们可以定义一个UserRepository接口继承通用的Repository接口。

```
public interface UserRepository extends Repository<User, Long> {
    User findByUsername(String username);
}
```

这使得UserRepository不仅继承了基础的CRUD操作，还可以扩展特定于用户的查询方法。

7.3.3 Repository 模式的实现逻辑

在使用Repository模式时，通常遵循以下设计逻辑：

1. 定义抽象Repository接口
，统一数据访问方法。
2. 创建具体的Repository接口
，根据业务需求添加个性化方法。
3. 实现Repository接口
，具体实现数据访问逻辑（JPA、MyBatis等）。
4. 在业务层使用Repository
，调用方法进行数据操作，而不直接操作数据库。

在此基础上，我们可以使用不同的数据访问技术来实现该模式，如Spring Data JPA和MyBatis。

7.4 在Spring Data JPA中的Repository实现

7.4.1 Spring Data JPA简介

Spring Data JPA 是 Spring 提供的一套基于 JPA 规范的持久层框架，它极大地简化了Repository的实现。

7.4.2 定义一个基础Repository接口


```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // 自动提供基本的CRUD方法
    User findByUsername(String username);
}
```

7.4.3 自定义查询

Spring Data JPA 支持使用方法名称派生查询，也可以使用 @Query 注解自定义SQL：

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.email = :email")
    User findByEmail(@Param("email") String email);
}
```

7.5 在MyBatis中实现Repository模式

7.5.1 使用Mapper接口

```
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

@Mapper
public interface UserMapper {
    @Select("SELECT * FROM users WHERE username = #{username}")
    User findByUsername(String username);
}
```

7.5.2 使用XML配置

```xml
<mapper namespace="com.example.mapper.UserMapper">
    <select id="findByUsername" resultType="com.example.entity.User">
        SELECT * FROM users WHERE username = #{username}
    </select>
</mapper>
```

7.6 造轮子：使用MyBatis实现Repository模式

如果没有Spring Data JPA，我们可以手动实现一个Repository。

7.6.1 定义Repository接口

```java
public interface UserRepository {
    User findByUsername(String username);
    void save(User user);
}
```

7.6.2 实现Repository


```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class UserRepositoryImpl implements UserRepository {
    @Autowired
    private UserMapper userMapper;

    @Override
    public User findByUsername(String username) {
        return userMapper.findByUsername(username);
    }

    @Override
    public void save(User user) {
        userMapper.insert(user);
    }
}
```

7.7 结论

Repository模式是一种强大的数据访问模式，在Spring Data JPA和MyBatis等框架中均有成熟的实现。它能够降低代码耦合，提高可维护性，使应用程序的数据访问更加清晰和可测试。

第八章：事务处理 & Unit Of Work

本章深入讲解事务处理的基本概念与原理，以及如何在 Java 和常用框架（如 Spring 和 Hibernate）中实现事务管理。同时，我们将介绍**Unit of Work**模式，解释它如何与事务处理结合使用，提升系统的一致性和可靠性。

8.1 事务的基本概念

事务（Transaction） 是数据库管理系统中对一组操作的封装，确保这些操作作为一个整体执行，要么全部成功，要么全部失败，从而保证数据的一致性、可靠性和完整性。

事务通常具备以下四个基本特性（ACID原则）：

- 原子性（Atomicity）：事务中的操作要么全部完成，要么全部不做，不能停留在中间状态。
- 一致性（Consistency）：事务的执行必须使数据库从一个一致性状态转变到另一个一致性状态。
- 隔离性（Isolation）：事务的执行不应受到其他事务的干扰，即事务的中间状态对其他事务是不可见的。
- 持久性（Durability）：一旦事务提交，其对数据库的修改应永久保存在数据库中，即使发生系统崩溃，也不会丢失。

8.1.1 事务的隔离级别

数据库提供了不同的隔离级别，以控制事务之间的可见性。常见的隔离级别有：

- READ UNCOMMITTED：最低的隔离级别，事务可以读取到其他事务未提交的数据（脏读）。
- READ COMMITTED：事务只能读取到已提交的数据，避免脏读，但可能出现不可重复读。
- REPEATABLE READ：事务中的读操作始终读取相同的数据，避免了脏读和不可重复读，但可能出现幻读。
- SERIALIZABLE：最高的隔离级别，事务之间完全隔离，防止脏读、不可重复读和幻读，但性能较差。

8.2 Java 中的事务处理

Java 提供了多种方式来管理事务，最常见的方式是通过 **JDBC** 和 **Spring**。

8.2.1 JDBC 中的事务管理

在 JDBC 中，事务通常通过 Connection 对象来管理。以下是 JDBC 中事务的常见操作：

```
Connection connection = dataSource.getConnection();
try {
    connection.setAutoCommit(false);  // 禁用自动提交
    // 执行多个 SQL 操作
    PreparedStatement stmt1 = connection.prepareStatement("UPDATE users SET balance = balance - 100 WHERE id = ?");
    stmt1.setInt(1, userId);
    stmt1.executeUpdate();

    PreparedStatement stmt2 = connection.prepareStatement("UPDATE accounts SET balance = balance + 100 WHERE id = ?");
    stmt2.setInt(1, recipientId);
    stmt2.executeUpdate();

    connection.commit();  // 提交事务
} catch (SQLException e) {
    connection.rollback();  // 回滚事务
} finally {
    connection.close();
}

```

在上述代码中，我们通过设置 connection.setAutoCommit(false) 来禁用自动提交，这样就可以显式地控制事务的提交或回滚。

8.2.2 Spring 中的事务管理

Spring 提供了更高层次的事务管理，它将事务与业务逻辑解耦，简化了开发。

- 声明式事务

：通过注解（@Transactional）或 XML 配置来管理事务。

```
@Transactional
public void transferMoney(int userId, int recipientId, double amount) {
    // 执行多个数据库操作
    userRepository.updateBalance(userId, -amount);
    accountRepository.updateBalance(recipientId, amount);
}

```

- 编程式事务

：通过 TransactionTemplate 或 PlatformTransactionManager 手动控制事务的开始、提交和回滚。

```
@Autowired
private PlatformTransactionManager transactionManager;

public void transferMoney(int userId, int recipientId, double amount) {
    TransactionStatus status = transactionManager.getTransaction(new
DefaultTransactionDefinition());
    try {
        // 执行操作
        userRepository.updateBalance(userId, -amount);
        accountRepository.updateBalance(recipientId, amount);
        transactionManager.commit(status);  // 提交事务
    } catch (Exception e) {
        transactionManager.rollback(status);  // 回滚事务
    }
}

```

8.3 Unit of Work 模式

Unit of Work 模式用于管理一组操作的事务，确保它们作为一个整体执行。它的核心思想是将多个操作作为一个“工作单元”来处理，直到所有操作都完成后，再决定是否提交或回滚。
在一个事务中，多个对象的操作可能会被记录到内存中，然后统一提交到数据库。Unit of Work 模式保证了数据库操作的一致性，并避免了重复更新数据库。

8.3.1 Unit of Work 的工作原理

- 开始工作单元：当业务逻辑执行时，首先创建一个 Unit of Work 对象，开始记录事务中的所有操作。
- 记录操作：对实体的新增、修改或删除操作被记录到工作单元中。
- 提交/回滚：当所有操作完成后，统一提交这些操作，或者在出现异常时回滚操作。

8.3.2 在 Spring 中使用 Unit of Work

Spring 中的 @Transactional 注解本身就实现了 Unit of Work 模式。所有在同一事务方法中执行的数据库操作都将在事务结束时统一提交或回滚。

例如，下面是一个简单的例子，展示了如何在 Spring 中使用 @Transactional 来实现 Unit of Work：


```
@Transactional
public void processOrder(Order order) {
    orderRepository.save(order);  // 保存订单
    paymentRepository.processPayment(order.getPayment());  // 处理支付
    inventoryRepository.updateInventory(order.getItems());  // 更新库存
}

```

在此示例中，processOrder 方法中的所有操作将在同一个事务中进行处理。只有在所有操作都成功完成时，事务才会提交，否则会回滚所有操作。

8.3.3 手动实现 Unit of Work

虽然 Spring 的声明式事务已经很好地支持了 Unit of Work 模式，但我们也可以手动实现这个模式。下面是一个简单的手动实现：

```
public class UnitOfWork {
    private List<Operation> operations = new ArrayList<>();

    public void registerOperation(Operation operation) {
        operations.add(operation);
    }

    public void commit() {
        for (Operation operation : operations) {
            operation.execute();
        }
    }

    public void rollback() {
        // 这里可以实现回滚逻辑
    }
}

```

这里的 Operation 是所有数据库操作的封装，commit() 方法会执行所有记录的操作，确保事务的一致性。

8.4 事务与 Unit of Work 在框架中的结合

在一些框架（如 Hibernate 和 Spring Data JPA）中，事务和 Unit of Work 模式通常结合使用。例如，Hibernate 的 Session 就是实现了 Unit of Work 模式的类，它会跟踪实体的状态变化并在事务提交时执行数据库操作。

在 Spring Data JPA 中，EntityManager 也充当了类似的角色，它会跟踪实体的变化，并在事务提交时将所有更改提交到数据库。

8.5 总结

- 事务是数据库管理中不可或缺的部分，它确保了数据的一致性和完整性。
- 在 Java 中，我们可以通过 JDBC 和 Spring 提供的事务管理方式来管理事务。
- Unit of Work 模式通过将多个操作作为一个工作单元来管理数据库操作，确保一致性。
- Spring 和 Hibernate 等框架通过实现 Unit of Work 模式，使得事务管理更加简洁和高效。

通过对事务管理和 Unit of Work 模式的深入理解，我们可以在实际项目中更好地确保数据一致性和操作的原子性，同时避免潜在的事务处理问题。

附录：手动实现 Unit of Work 模式

Unit of Work 模式的核心是把一组数据库操作（如插入、更新、删除）包装在一个“工作单元”中，直到所有操作完成后，统一提交或回滚。这样，我们可以确保一组操作的原子性，避免中间状态对外部系统的影响。

下面将通过一个简单的手动实现，逐步解释 **Unit of Work** 模式的工作原理。

---

1. 定义基本接口

首先，我们需要定义一个 Operation 接口，用于表示每个操作。这个接口包括一个 execute() 方法，所有数据库操作都应该实现这个接口。

```
// 操作接口：每个操作都需要实现此接口
public interface Operation {
    void execute();  // 执行操作
}

```

---

2. 定义具体的操作类

接下来，我们可以为每种操作（比如 INSERT、UPDATE 和 DELETE）实现 Operation 接口。这样，我们就能将不同类型的操作封装为对象，并统一管理它们。

```
// 插入操作
```



```
public class InsertOperation implements Operation {
    private Entity entity;

    public InsertOperation(Entity entity) {
        this.entity = entity;
    }

    @Override
    public void execute() {
        // 执行数据库插入操作
        System.out.println("Inserting entity: " + entity);
        // 这里实际会调用数据库操作，如 JdbcTemplate.insert()
    }
}

// 更新操作
public class UpdateOperation implements Operation {
    private Entity entity;

    public UpdateOperation(Entity entity) {
        this.entity = entity;
    }

    @Override
    public void execute() {
        // 执行数据库更新操作
        System.out.println("Updating entity: " + entity);
        // 这里实际会调用数据库操作，如 JdbcTemplate.update()
    }
}

// 删除操作
public class DeleteOperation implements Operation {
    private Entity entity;

    public DeleteOperation(Entity entity) {
        this.entity = entity;
    }
```

```
    @Override
    public void execute() {
        // 执行数据库删除操作
        System.out.println("Deleting entity: " + entity);
        // 这里实际会调用数据库操作，如 JdbcTemplate.delete()
    }
}
```

在上述代码中，我们为 INSERT、UPDATE 和 DELETE 操作分别创建了三个具体类 InsertOperation、UpdateOperation 和 DeleteOperation，每个类的 execute() 方法都代表了对数据库的实际操作。

---

3. 定义 Unit of Work 类

UnitOfWork 类是 **Unit of Work** 模式的核心，它负责管理所有操作，并在事务结束时统一提交或回滚。这些操作将被统一记录，然后按顺序执行。


```
import java.util.ArrayList;
import java.util.List;

// Unit of Work 管理类
public class UnitOfWork {
    private List<Operation> operations = new ArrayList<>();

    // 注册操作
    public void registerOperation(Operation operation) {
        operations.add(operation);  // 将操作加入待执行列表
    }

    // 提交操作
    public void commit() {
        System.out.println("Committing transaction...");
        for (Operation operation : operations) {
            operation.execute();  // 执行所有操作
        }
    }

    // 回滚操作
    public void rollback() {
        System.out.println("Rolling back transaction...");
        // 回滚操作的具体实现可以根据需求进行处理
        operations.clear();  // 清空待执行操作
    }
}
```

UnitOfWork 类有两个核心方法：

- registerOperation()：用于注册操作，记录所有要执行的操作。
- commit()：用于提交操作，执行所有注册的操作。
- rollback()：用于回滚操作，可以清空所有操作，或者做一些特定的回滚逻辑。

---

4. 使用 Unit of Work

最后，我们需要创建一个简单的应用来演示如何使用 UnitOfWork 模式。假设我们有一个 Entity 类，并通过手动操作来管理事务。

```
// 简单的实体类
```



```
public class Entity {
    private String name;

    public Entity(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Entity{name='" + name + "'}";
    }
}

public class UnitOfWorkExample {
    public static void main(String[] args) {
        // 创建 UnitOfWork 实例
        UnitOfWork unitOfWork = new UnitOfWork();

        // 创建实体对象
        Entity entity1 = new Entity("Entity 1");
        Entity entity2 = new Entity("Entity 2");

        // 创建操作
        Operation insertOp = new InsertOperation(entity1);
        Operation updateOp = new UpdateOperation(entity2);

        // 注册操作到 UnitOfWork
        unitOfWork.registerOperation(insertOp);
        unitOfWork.registerOperation(updateOp);

        // 提交操作
        try {
            unitOfWork.commit();  // 提交事务
        } catch (Exception e) {
            unitOfWork.rollback();  // 如果出现异常，回滚事务
        }
    }
}
```

5. 解释

1. 实体类：我们定义了一个 Entity 类来代表数据库中的实体。在实际项目中，Entity 可能会是 JPA 或 Hibernate 实体。
2. 操作注册：在 UnitOfWorkExample 类中，我们创建了两个操作 insertOp 和 updateOp，并将它们注册到 UnitOfWork 中。这些操作会被 UnitOfWork 对象跟踪。
3. 事务提交：调用 unitOfWork.commit() 方法将所有操作一次性提交。这时，所有数据库操作会依次执行。
4. 回滚：如果发生异常，我们可以调用 unitOfWork.rollback() 来回滚所有操作。这里的回滚操作比较简单，但在实际应用中，可以根据需要添加更多的回滚策略，比如撤销数据库修改。

---

6. 扩展：结合数据库连接池和事务

在实际开发中，UnitOfWork 模式通常与数据库事务和连接池结合使用。例如，我们可以在 UnitOfWork 中管理一个数据库连接，并在事务提交时处理所有 SQL 操作。这样，UnitOfWork 可以不仅仅是逻辑的封装，还能控制实际的数据库连接和事务管理。

例如，可以结合 Spring 或 JDBC 连接池来管理数据库连接，在事务的开始时从连接池中获取一个连接，在 commit() 时提交事务，rollback() 时回滚事务。

总结

通过手动实现 **Unit of Work** 模式，我们将数据库操作封装在一个“工作单元”中，确保这些操作要么全部成功，要么全部失败。这种模式帮助我们管理和控制一系列数据库操作，避免出现中间状态，从而保证数据的一致性和原子性。

在实际应用中，像 Hibernate、JPA、Spring 等框架都已经内置了类似的功能，但手动实现 **Unit of Work** 模式能够让我们深入理解事务管理的本质及其如何帮助我们在复杂的系统中保持一致性。

第九章：非关系型数据处理

在传统的关系型数据库中，数据以表格形式组织，具有严格的结构和约束。然而，在现代应用中，越来越多的数据并不完全符合关系模型，尤其是当数据的结构不固定或需要高可扩展性时，非关系型数据库（NoSQL）提供了更为灵活和高效的解决方案。本章将介绍如何处理非关系型数据，重点讲解常用的 NoSQL 数据库及其在 Spring 和 Java 环境中的应用。

9.1 NoSQL 数据库概述

NoSQL（Not Only SQL）是指一类不使用传统关系模型的数据库系统。它通常用于处理大规模、非结构化、半结构化的数据，特别适用于分布式存储和高并发的应用场景。与关系型数据库相比，NoSQL 数据库具有以下特点：

- 灵活的架构：没有固定的表结构，可以处理不规则的数据。
- 高可扩展性：支持水平扩展，通过增加更多节点来处理更多的数据。
- 高性能：可以优化数据访问和写入速度，适合大数据量和高并发的场景。
- 分布式存储：天然支持分布式架构，数据可以分布在多个节点上。

常见的 NoSQL 数据库类型：

- 键值存储（Key-Value Stores）：如 **Redis** 和 **Riak**，数据以键值对形式存储，适合缓存等场景。
- 文档存储（Document Stores）：如 **MongoDB** 和 **CouchDB**，数据以文档形式存储，支持灵活的查询和索引，适合存储 JSON 或 BSON 格式的数据。
- 列族存储（Column Family Stores）：如 **HBase** 和 **Cassandra**，将数据按列簇存储，适合大数据处理。
- 图数据库（Graph Databases）：如 **Neo4j**，用于存储和处理复杂的关系数据，适用于社交网络、推荐引擎等场景。

9.2 MongoDB —— 文档存储数据库

9.2.1 MongoDB 简介

MongoDB 是一种面向文档的 NoSQL 数据库，它存储的数据以 BSON（类似 JSON）格式存储，而不是传统的行和列形式。MongoDB 的主要特点包括：

- 数据以文档形式存储，文档是一个键值对的集合，支持复杂的数据类型，如数组、嵌套文档等。
- 灵活的模式：不需要预先定义数据库模式，可以存储不同结构的数据。
- 支持丰富的查询功能，包括嵌套查询、聚合、索引等。

9.2.2 使用 MongoDB

在 Spring 环境中使用 MongoDB，最常见的方式是通过 Spring Data MongoDB 来实现。Spring Data MongoDB 提供了便捷的接口和注解，使得我们可以轻松地与 MongoDB 进行交互。

连接 MongoDB

首先，我们需要在 pom.xml 中添加 MongoDB 的依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>

```

然后，在 application.properties 或 application.yml 中配置 MongoDB 的连接信息：

```
spring.data.mongodb.uri=mongodb://localhost:27017/mydatabase

```

创建 MongoDB 实体

在 MongoDB 中，我们使用 Java 类表示文档，通过注解 @Document 来定义实体：

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "users")  // 指定 MongoDB 中的集合名称
public class User {

    @Id  // 用于标识文档的主键
    private String id;
    private String name;
    private int age;

    // 构造方法、getter 和 setter
}

```

使用 MongoRepository 进行数据操作

Spring Data MongoDB 提供了 MongoRepository 接口来简化数据库操作。通过继承 MongoRepository，我们可以自动获得增、删、改、查等常用方法：

```
import org.springframework.data.mongodb.repository.MongoRepository;

public interface UserRepository extends MongoRepository<User, String> {
    // 可以定义自定义查询方法，如通过名称查找用户
    List<User> findByName(String name);
}

```

操作数据

在服务层，我们可以注入 UserRepository 来进行数据库操作：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public void createUser(User user) {
        userRepository.save(user);  // 保存用户
    }

    public List<User> getUsersByName(String name) {
        return userRepository.findByName(name);  // 查询用户
    }
}

```

聚合操作

MongoDB 提供了强大的聚合框架，支持复杂的数据聚合操作，如分组、排序、求和等。在 Spring Data MongoDB 中，聚合操作可以通过 Aggregation 类来实现：

```java
import org.springframework.data.mongodb.core.aggregation.Aggregation;
import org.springframework.data.mongodb.core.aggregation.AggregationResults;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;

@Service
public class UserService {

    @Autowired
    private MongoTemplate mongoTemplate;

    public void aggregateUsers() {
        Aggregation aggregation = Aggregation.newAggregation(
            Aggregation.group("age").count().as("count")  // 按年龄分组并统计人数
        );

        AggregationResults<User> results = mongoTemplate.aggregate(aggregation, "users", User.class);
        List<User> aggregatedResults = results.getMappedResults();
    }
}

```

9.3 Redis —— 键值存储数据库

9.3.1 Redis 简介

Redis 是一个高性能的键值对存储数据库，它不仅支持简单的键值对存储，还支持字符串、列表、集合、有序集合、哈希等多种数据结构。Redis 主要用于缓存、会话存储、排行榜等场景，适合高并发和低延迟的应用。

9.3.2 使用 Redis

Spring 提供了对 Redis 的良好支持，可以通过 Spring Data Redis 来简化与 Redis 的交互。

配置 Redis

首先，在 pom.xml 中添加 Redis 的依赖：


```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

```

在 application.properties 中配置 Redis 连接信息：

```properties
spring.redis.host=localhost
spring.redis.port=6379

```

使用 RedisTemplate

Spring Data Redis 提供了 RedisTemplate 用于与 Redis 进行交互。下面是一个基本的使用示例：

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Service;

@Service
public class RedisService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    public void saveData(String key, String value) {
        redisTemplate.opsForValue().set(key, value);  // 设置键值对
    }

    public String getData(String key) {
        return redisTemplate.opsForValue().get(key);  // 获取键值
    }
}

```

9.4 NoSQL 数据与关系型数据的混合应用

在实际应用中，许多系统会同时使用关系型数据库和非关系型数据库。关系型数据库用于存储结构化数据，而 NoSQL 数据库用于存储非结构化或半结构化数据。

例如，常见的场景包括：

- 使用 **MongoDB** 存储用户的日志和社交网络数据。
- 使用 **Redis** 作为缓存存储，减少对关系型数据库的查询压力。

这种混合应用的方式能够充分发挥两种数据库的优势，提高系统的灵活性、可扩展性和性能。

9.5 总结

本章介绍了非关系型数据库（NoSQL）的基本概念，并深入探讨了 MongoDB 和 Redis 两种常见的 NoSQL 数据库的使用。在 Spring 环境下，我们可以使用 **Spring Data MongoDB** 和 **Spring Data Redis** 来简化与这些数据库的交互。通过灵活选择数据库类型和合理使用，可以大大提高系统的性能和可扩展性。在实际应用中，结合使用关系型数据库和 NoSQL 数据库，可以充分利用它们的各自优势，满足多样化的数据存储需求。

HQL（Hibernate Query Language）是 Hibernate 提供的一种面向对象的查询语言，它的语法类似于 SQL，但操作的是对象和属性，而不是数据库表和列。HQL 是 Hibernate 的核心特性之一，能够帮助开发者以面向对象的方式编写查询语句。以下是 HQL 的详细语法介绍：

---

1. HQL 基本语法

HQL 的基本语法结构与 SQL 类似，主要包括以下几个部分：

- SELECT：指定查询的返回结果。
- FROM：指定查询的实体类。
- WHERE：指定查询条件。
- ORDER BY：指定排序规则。
- GROUP BY：指定分组规则。
- HAVING：指定分组后的过滤条件。

1.1 基本查询

```
FROM EntityName
```

- 查询所有实体对象。
  示例：
-

```
FROM User
```

等价于 SQL：

```
SELECT * FROM users;
```

1.2 指定返回字段

```
SELECT property1, property2 FROM EntityName
```

- 查询指定字段。
  示例：
-

```
SELECT name, email FROM User
```

等价于 SQL：

```
SELECT name, email FROM users;
```

1.3 条件查询

```
FROM EntityName WHERE condition
```

- 根据条件过滤结果。
- 示例：

```
FROM User WHERE name = 'Alice'
```

等价于 SQL：

```
SELECT * FROM users WHERE name = 'Alice';
```

1.4 排序

```
FROM EntityName ORDER BY property ASC|DESC
```

- 按指定字段排序。
- 示例：

```
FROM User ORDER BY name ASC
```

等价于 SQL：


```
SELECT * FROM users ORDER BY name ASC;
```

1.5 分页查询

HQL 本身不支持分页语法，但可以通过 Query 接口实现分页：

```
String hql = "FROM User";
Query<User> query = session.createQuery(hql, User.class);
query.setFirstResult(0); // 起始位置
query.setMaxResults(10); // 每页大小
List<User> users = query.getResultList();
```

---

2. HQL 高级语法

2.1 聚合函数

HQL 支持常见的聚合函数，如 COUNT、SUM、AVG、MIN、MAX。

- 示例：

```
SELECT COUNT(*) FROM User
```

等价于 SQL：

```
SELECT COUNT(*) FROM users;
```

2.2 分组查询

```
SELECT property, aggregate_function(property)
FROM EntityName
GROUP BY property
```

- 示例：

```
SELECT department, COUNT(*) FROM Employee GROUP BY department
```

等价于 SQL：

```
SELECT department, COUNT(*) FROM employees GROUP BY department;
```

2.3 连接查询

HQL 支持内连接、左外连接和右外连接。

- 内连接：

```
FROM Entity1 e JOIN e.entity2
```

示例：

```
FROM Order o JOIN o.customer
```

等价于 SQL：

```
SELECT * FROM orders o INNER JOIN customers c ON o.customer_id = c.id;
```

- 左外连接：

```
FROM Entity1 e LEFT JOIN e.entity2
```

示例：

```
FROM Order o LEFT JOIN o.customer
```

等价于 SQL：


```
SELECT * FROM orders o LEFT JOIN customers c ON o.customer_id = c.id;
```

2.4 子查询

HQL 支持子查询，子查询必须放在括号中。

- 示例：

```
FROM User WHERE id IN (SELECT userId FROM Order WHERE amount > 100)
```

等价于 SQL：

```
SELECT * FROM users WHERE id IN (SELECT user_id FROM orders WHERE amount > 100);
```

---

3. HQL 参数绑定

HQL 支持两种参数绑定方式：**位置参数**和**命名参数**。

3.1 位置参数

使用 ? 占位符，参数按顺序绑定。

- 示例：

```
FROM User WHERE name = ? AND email = ?
```

Java 代码：

```
Query<User> query = session.createQuery("FROM User WHERE name = ? AND email = ?",
User.class);
query.setParameter(0, "Alice");
query.setParameter(1, "alice@example.com");
List<User> users = query.getResultList();
```

3.2 命名参数

使用 :paramName 占位符，参数按名称绑定。

- 示例：

```
FROM User WHERE name = :name AND email = :email
```

Java 代码：

```
Query<User> query = session.createQuery("FROM User WHERE name = :name AND email = :email", User.class);
query.setParameter("name", "Alice");
query.setParameter("email", "alice@example.com");
List<User> users = query.getResultList();
```

---

4. HQL 函数

HQL 提供了一些内置函数，可以用于查询中：

- 字符串函数：CONCAT、SUBSTRING、LOWER、UPPER、TRIM。
- 数学函数：ABS、SQRT、MOD。
- 时间函数：CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP。

示例：

```
SELECT CONCAT(name, ' - ', email) FROM User
```

等价于 SQL：

```
SELECT CONCAT(name, ' - ', email) FROM users;
```

---

5. HQL 查询示例

5.1 查询所有用户


```
FROM User
```

5.2 查询指定字段

```
SELECT name, email FROM User
```

5.3 条件查询

```
FROM User WHERE age > 18 AND email LIKE '%@example.com'
```

5.4 聚合查询

```
SELECT COUNT(*) FROM User WHERE age > 18
```

5.5 连接查询

```
FROM Order o JOIN o.customer WHERE o.amount > 100
```

5.6 子查询

```
FROM User WHERE id IN (SELECT userId FROM Order WHERE amount > 100)
```

---

6. 总结

HQL 是 Hibernate 提供的面向对象查询语言，语法与 SQL 类似，但操作的是对象和属性。HQL 支持基本查询、条件查询、聚合查询、连接查询和子查询等高级功能。通过参数绑定和内置函数，HQL 可以满足复杂的查询需求。掌握 HQL 是使用 Hibernate 的关键技能之一。

JPA（Java Persistence API）标准 是由**Java Community Process (JCP)** 制定的，用于在Java平台上处理对象关系映射（ORM）的一种标准。
它定义了一套规范，使得Java开发者可以在不同的持久化框架（如Hibernate、EclipseLink、OpenJPA等）之间实现一致的持久化操作，从而避免了与具体框架绑定的代码，提供了更好的跨平台兼容性。

JPA标准的主要内容：

1. 对象与关系数据库表的映射：
   - JPA定义了如何将Java对象映射到数据库表中。通过注解或XML配置，可以定义类与表、字段与列之间的映射关系。
   - @Entity: 标记一个类为实体类，映射到数据库的一个表。
   - @Table: 可以指定实体类映射的表名。
   - @Id: 标记主键字段。
   - @Column: 映射类的属性与数据库表的列。
2. 持久化上下文：
   - JPA定义了持久化上下文（Persistence Context），它是一个运行时环境，负责管理实体对象的生命周期。它可以通过**EntityManager**进行访问，负责从数据库加载、保存、删除实体对象。
3. 查询语言（JPQL）：
   - JPA提供了**JPQL (Java Persistence Query Language)**，一种基于对象的查询语言。与SQL不同，JPQL操作的是实体类对象，而不是数据库表。它允许开发者在不关心底层数据库的情况下，进行数据库查询。

例如：

```
TypedQuery<User> query = entityManager.createQuery("SELECT u FROM User u WHERE u.age > 30", User.class);
List<User> results = query.getResultList();

```

4. 事务管理：
   - JPA通过**EntityTransaction**接口管理事务，支持与Java EE的事务管理进行集成。
   - 通过JPA，开发者可以通过容器管理的事务或应用程序控制事务来操作数据库。
5. 生命周期管理：
   - JPA定义了实体对象的生命周期，实体对象可以处于以下几种状态：
     - Transient: 未持久化的对象，尚未与数据库进行关联。
     - Managed: 被持久化上下文管理的对象，可以与数据库同步。
     - Detached: 从持久化上下文分离的对象，不能自动同步到数据库。

     - Removed: 被删除的对象，将会从数据库中移除。
6. 关系映射：
   - 一对多（OneToMany）、**多对一（ManyToOne）**、**一对一（OneToOne）**、**多对多（ManyToMany）**等关系类型的映射。
   - JPA提供了注解来定义这些关系，如**@ManyToOne**、**@OneToMany**、**@ManyToMany**等。
7. 级联操作：
   - JPA允许级联操作，即对一个实体的操作可以传播到相关联的实体。
   - 例如，通过设置**CascadeType.ALL**，保存一个实体时，相关的其他实体也会被自动保存。
   - 级联操作包括：**Persist**、**Merge**、**Remove**、**Refresh**等。
8. 实体图（Entity Graph）：
   - JPA提供了实体图的功能，用于控制查询的加载策略，支持**懒加载**和**急加载**（Lazy Loading 和 Eager Loading）。
9. 缓存管理：
   - JPA提供了第一层缓存（EntityManager所在的持久化上下文的缓存）和第二层缓存（跨多个会话的缓存）。第二层缓存是可选的，由具体的实现（如Hibernate）支持。

JPA标准的优势：

- 数据库无关性：使用JPA可以使应用程序代码与底层数据库解耦，能够在不同的数据库间切换，增加了应用的可移植性。
- 减少开发工作量：JPA自动生成SQL、管理对象生命周期、处理事务等，减少了大量手写SQL和数据库连接管理的工作。
- 集成框架支持：JPA与Spring等流行框架兼容，能够轻松集成到各种Java EE和Spring项目中。
- 标准化：作为Java官方的标准，JPA为ORM操作提供了一个统一的规范，减少了不同框架之间的差异。

JPA的局限性：

- 性能开销：尽管JPA自动处理了很多复杂的数据库操作，但它也可能带来性能开销，尤其是在涉及大量数据或复杂查询时。
- 配置复杂性：JPA的配置相比于其他一些轻量级的ORM框架（如MyBatis）更复杂。
- 灵活性不足：对于一些复杂的查询和数据库操作，JPA的标准功能可能不够灵活，开发者有时需要依赖JPA实现的具体功能（如Hibernate）。

总结：

JPA标准通过定义ORM的基本规范，使得Java开发者能够以标准化的方式与数据库交互，并提供了对数据库操作的抽象和自动化管理。虽然JPA本身并不实现ORM功能，但它作为规范，与具体实现（如Hibernate、EclipseLink）一起提供了强大的ORM功能，广泛应用于企业级应用程序中。

Spring Data JPA 功能全面覆盖

1. Repository 接口

Spring Data JPA 提供了多种 Repository 接口，开发者可以根据需求选择：

- CrudRepository：提供基本的 CRUD 操作。
- PagingAndSortingRepository：在 CrudRepository 的基础上增加了分页和排序功能。
- JpaRepository：最常用的接口，继承了 CrudRepository 和 PagingAndSortingRepository，并增加了 JPA 特定的功能（如批量删除）。

```
public interface UserRepository extends JpaRepository<User, Long> {
    // 自定义方法
}
```

---

2. 方法名自动生成查询

Spring Data JPA 支持根据方法名自动生成查询，无需编写 SQL 或 JPQL。规则如下：

- 基本查询：findBy、readBy、queryBy、getBy。
- 条件查询：findByNameAndEmail、findByAgeGreaterThan 等。
- 排序和分页：findAll(Sort sort)、findAll(Pageable pageable)。

```
List<User> findByName(String name);
List<User> findByNameAndEmail(String name, String email);
Page<User> findByAgeGreaterThan(int age, Pageable pageable);
```

---

3. 自定义查询

通过 @Query 注解，开发者可以定义复杂的 JPQL 或原生 SQL 查询。

3.1 JPQL 查询

```
@Query("SELECT u FROM User u WHERE u.email LIKE %:email%")
List<User> findByEmailContaining(@Param("email") String email);
```

3.2 原生 SQL 查询

```
@Query(value = "SELECT * FROM users WHERE email LIKE %:email%", nativeQuery = true)
List<User> findByEmailContainingNative(@Param("email") String email);
```

3.3 修改查询

通过 @Modifying 注解标记修改操作（如 UPDATE、DELETE）。

```
@Modifying
@Query("UPDATE User u SET u.name = :name WHERE u.id = :id")
int updateUserName(@Param("id") Long id, @Param("name") String name);
```

---

4. 分页和排序

Spring Data JPA 提供了对分页和排序的内置支持。

4.1 分页查询

```
Page<User> findAll(Pageable pageable);
```

4.2 排序查询

```
List<User> findAll(Sort sort);
```

4.3 分页和排序结合

```
Page<User> findByName(String name, Pageable pageable);
```

---

5. 动态查询

Spring Data JPA 支持通过 Specification 和 Querydsl 实现动态查询。

5.1 使用 Specification


```
public class UserSpecifications {
    public static Specification<User> nameLike(String name) {
        return (root, query, criteriaBuilder) ->
criteriaBuilder.like(root.get("name"), "%" + name + "%");
    }
}

List<User> users = userRepository.findAll(UserSpecifications.nameLike("Alice"));
```

5.2 使用 Querydsl

Querydsl 提供了类型安全的查询方式，适合复杂查询场景。

```
QUser user = QUser.user;
List<User> users = userRepository.findAll(user.name.eq("Alice"));
```

---

6. 事务管理

Spring Data JPA 与 Spring 的事务管理无缝集成，支持声明式事务管理。

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    @Transactional
    public void updateUser(Long id, String name) {
        User user = userRepository.findById(id).orElseThrow(() -> new
RuntimeException("User not found"));
        user.setName(name);
        userRepository.save(user);
    }
}
```

---

7. 审计功能

Spring Data JPA 提供了审计功能，可以自动记录实体的创建时间、修改时间等。

7.1 启用审计

```
@Configuration
@Enableable JpaAuditing
pul lic class JpaConfig
    { @Bean
    public AuditorAware<String> auditorAware() {
        return () -> Optional.of("System"); // 返回当前用户或系统标识
    }
}
```

7.2 实体类配置

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;
}
```

---

8. 投影（Projection）

Spring Data JPA 支持投影功能，允许只查询部分字段。

8.1 接口投影

```java
public interface UserNameOnly {
    String getName();
}

List<UserNameOnly> findByName(String name);
```

8.2 类投影

```java
public class UserDTO {
    private String name;
    private String email;

    public UserDTO(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and Setters
}

@Query("SELECT new com.example.UserDTO(u.name, u.email) FROM User u WHERE u.name = :name")
List<UserDTO> findUserDTOByName(@Param("name") String name);
```

---

9. 多数据源支持

Spring Data JPA 支持配置多个数据源，适用于需要连接多个数据库的场景。

9.1 配置多数据源


```java
@Configuration
@EnableJpaRepositories(
    basePackages = "com.example.primary",
    entityManagerFactoryRef = "primaryEntityManagerFactory",
    transactionManagerRef = "primaryTransactionManager"
)
public class PrimaryDataSourceConfig {
    // 配置主数据源
}

@Configuration
@EnableJpaRepositories(
    basePackages = "com.example.secondary",
    entityManagerFactoryRef = "secondaryEntityManagerFactory",
    transactionManagerRef = "secondaryTransactionManager"
)
public class SecondaryDataSourceConfig {
    // 配置从数据源
}
```

---

10. 总结

Spring Data JPA 的功能非常丰富，涵盖了从基础的 CRUD 操作到高级的动态查询、审计、投影等特性。通过合理使用这些功能，开发者可以极大地简化数据访问层的开发，并提高代码的可维护性和性能。

1. 方法名自动生成查询的原理

Spring Data JPA 通过解析方法名，自动推导出查询的逻辑。方法名的命名规则由以下几部分组成：

- 前缀：如 findBy、readBy、queryBy、getBy 等。
- 属性名：实体类的属性名，首字母大写。
- 条件关键字：如 And、Or、Between、LessThan 等。
- 参数：方法的参数列表，与属性名和条件关键字对应。

Spring Data JPA 会根据方法名生成相应的 JPQL 查询，并自动处理参数绑定。

---

2. 方法名命名规则

以下是一些常见的方法名命名规则及其对应的查询逻辑：

2.1 简单条件查询

- findBy属性名：根据某个属性查询。

```
List<User> findByName(String name);
```

生成的查询：

```
SELECT u FROM User u WHERE u.name = ?1
```

- findBy属性名1And属性名2：根据多个属性组合查询。

```
List<User> findByNameAndEmail(String name, String email);
```

生成的查询：

```
SELECT u FROM User u WHERE u.name = ?1 AND u.email = ?2
```

- findBy属性名1Or属性名2：根据多个属性组合查询（或条件）。

```
List<User> findByNameOrEmail(String name, String email);
```


生成的查询：

```
SELECT u FROM User u WHERE u.name = ?1 OR u.email = ?2
```

2.2 比较条件查询

- findBy属性名GreaterThan：查询属性值大于指定值的记录。

```
List<User> findByAgeGreaterThan(int age);
```

生成的查询：

```
SELECT u FROM User u WHERE u.age > ?1
```

- findBy属性名LessThan：查询属性值小于指定值的记录。

```
List<User> findByAgeLessThan(int age);
```

生成的查询：

```
SELECT u FROM User u WHERE u.age < ?1
```

- findBy属性名Between：查询属性值在指定范围内的记录。

```
List<User> findByAgeBetween(int startAge, int endAge);
```

生成的查询：

```
SELECT u FROM User u WHERE u.age BETWEEN ?1 AND ?2
```

2.3 模糊查询

- findBy属性名Like：查询属性值匹配指定模式的记录。

```
List<User> findByNameLike(String namePattern);
```

生成的查询：

```
SELECT u FROM User u WHERE u.name LIKE ?1
```

2.4 排序查询

- findBy属性名OrderBy属性名Asc：查询并按照指定属性升序排序。

```
List<User> findByNameOrderByAgeAsc(String name);
```

生成的查询：

```
SELECT u FROM User u WHERE u.name = ?1 ORDER BY u.age ASC
```

- findBy属性名OrderBy属性名Desc：查询并按照指定属性降序排序。

```
List<User> findByNameOrderByAgeDesc(String name);
```

生成的查询：

```
SELECT u FROM User u WHERE u.name = ?1 ORDER BY u.age DESC
```

2.5 分页查询

- findBy属性名：结合 Pageable 参数实现分页查询。


```
Page<User> findByName(String name, Pageable pageable);
```

生成的查询：

```
SELECT u FROM User u WHERE u.name = ?1
```

---

3. 方法名自动生成查询的示例

以下是一个完整的示例，展示了如何使用方法名自动生成查询：

3.1 实体类

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
    private int age;

    // Getters and Setters
}
```

3.2 Repository 接口

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

import java.util.List;

public interface UserRepository extends JpaRepository<User, Long> {
    // 根据名称查询
    List<User> findByName(String name);

    // 根据名称和邮箱查询
    List<User> findByNameAndEmail(String name, String email);

    // 根据年龄大于指定值查询
    List<User> findByAgeGreaterThan(int age);

    // 根据名称模糊查询
    List<User> findByNameLike(String namePattern);

    // 根据名称查询并按照年龄升序排序
    List<User> findByNameOrderByAgeAsc(String name);

    // 根据名称分页查询
    Page<User> findByName(String name, Pageable pageable);
}
```

3.3 使用 Repository


```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;

import java.util.List;

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public List<User> findUsersByName(String name) {
        return userRepository.findByName(name);
    }

    public List<User> findUsersByNameAndEmail(String name, String email) {
        return userRepository.findByNameAndEmail(name, email);
    }

    public List<User> findUsersByAgeGreaterThan(int age) {
        return userRepository.findByAgeGreaterThan(age);
    }

    public List<User> findUsersByNameLike(String namePattern) {
        return userRepository.findByNameLike(namePattern);
    }

    public List<User> findUsersByNameOrderByAgeAsc(String name) {
        return userRepository.findByNameOrderByAgeAsc(name);
    }

    public Page<User> findUsersByNameWithPagination(String name, int page, int size) {
        return userRepository.findByName(name, PageRequest.of(page, size));
    }
}
```

4. 方法名自动生成查询的注意事项

1. 属性名必须正确：方法名中的属性名必须与实体类的属性名一致（忽略大小写）。
2. 参数顺序必须匹配：方法参数必须与条件关键字对应的属性顺序一致。
3. 复杂查询不适用：对于复杂的查询逻辑（如多表连接、子查询等），建议使用 @Query 注解或 Specification。
4. 性能问题：自动生成的查询可能不够高效，尤其是在处理大数据量时，建议结合索引和查询优化。

---

5. 总结

方法名自动生成查询是 Spring Data JPA 的一个强大特性，能够极大地简化数据访问层的开发。通过遵循命名规则，开发者可以快速定义查询方法，而无需编写复杂的 SQL 或 JPQL。然而，对于复杂的查询逻辑，仍需结合 @Query 注解或其他高级功能来实现。

分页和排序 是 Spring Data JPA 中非常实用的功能，特别适合处理大量数据时的分页显示和排序需求。Spring Data JPA 通过 Pageable 和 Sort 参数，提供了对分页和排序的内置支持。以下是一个详细的示例说明：

---

1. 分页和排序的核心类

- Pageable：用于分页查询，包含分页信息（如页码、每页大小）和排序信息。
- Page：分页查询的结果，包含数据列表、总页数、总记录数等信息。
- Sort：用于排序查询，指定排序字段和排序方向（升序或降序）。

---

2. 分页和排序的示例

2.1 实体类

假设我们有一个 User 实体类：

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
    private int age;

    // Getters and Setters
}
```

2.2 Repository 接口

在 Repository 接口中，定义一个支持分页和排序的查询方法：

```java
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // 分页查询所有用户
    Page<User> findAll(Pageable pageable);

    // 根据名称分页查询用户
    Page<User> findByName(String name, Pageable pageable);

    // 根据年龄分页查询用户，并按照名称排序
    Page<User> findByAge(int age, Pageable pageable);
}
```

2.3 使用分页和排序

在 Service 层中，使用 Pageable 和 Sort 实现分页和排序。


```java
import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    /**
     * 分页查询所有用户
     *
     * @param page 页码（从 0 开始）
     * @param size 每页大小
     * @return 分页结果
     */
    public Page<User> getAllUsers(int page, int size) {
        Pageable pageable = PageRequest.of(page, size);
        return userRepository.findAll(pageable);
    }

    /**
     * 根据名称分页查询用户
     *
     * @param name 名称
     * @param page 页码（从 0 开始）
     * @param size 每页大小
     * @return 分页结果
     */
    public Page<User> getUsersByName(String name, int page, int size) {
        Pageable pageable = PageRequest.of(page, size);
        return userRepository.findByName(name, pageable);
    }

    /**
     * 根据年龄分页查询用户，并按照名称升序排序
     *
     * @param age  年龄
     * @param page 页码（从 0 开始）
     * @param size 每页大小
     * @return 分页结果
     */
    public Page<User> getUsersByAgeWithSort(int age, int page, int size) {
        Sort sort = Sort.by(Sort.Direction.ASC, "name"); // 按照名称升序排序
        Pageable pageable = PageRequest.of(page, size, sort);
        return userRepository.findByAge(age, pageable);
    }

    /**
     * 根据年龄分页查询用户，并按照多个字段排序
     *
     * @param age  年龄
     * @param page 页码（从 0 开始）
     * @param size 每页大小
     * @return 分页结果
     */
    public Page<User> getUsersByAgeWithMultiSort(int age, int page, int size) {
        Sort sort = Sort.by(Sort.Direction.ASC, "name") // 按照名称升序排序
                        .and(Sort.by(Sort.Direction.DESC, "age")); // 按照年龄降序排序
        Pageable pageable = PageRequest.of(page, size, sort);
        return userRepository.findByAge(age, pageable);
    }
}
```

2.4 控制器层（Controller）

在控制器层中调用 Service 方法，并返回分页结果。

```
import org.springframework.beans.factory.annotation.Autowired;
```



```
import org.springframework.data.domain.Page;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping
    public Page<User> getAllUsers(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size) {
        return userService.getAllUsers(page, size);
    }

    @GetMapping("/search")
    public Page<User> getUsersByName(
            @RequestParam String name,
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size) {
        return userService.getUsersByName(name, page, size);
    }

    @GetMapping("/age")
    public Page<User> getUsersByAgeWithSort(
            @RequestParam int age,
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size) {
        return userService.getUsersByAgeWithSort(age, page, size);
    }

    @GetMapping("/age/multi-sort")
    public Page<User> getUsersByAgeWithMultiSort(
            @RequestParam int age,
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size) {
        return userService.getUsersByAgeWithMultiSort(age, page, size);
    }
```

```
}
```

---

3. 分页和排序的结果

3.1 分页结果（Page 对象）

Page 对象包含以下信息：

- content：当前页的数据列表。
- totalPages：总页数。
- totalElements：总记录数。
- number：当前页码（从 0 开始）。
- size：每页大小。
- sort：排序信息。

3.2 示例输出

假设数据库中有 100 条用户记录，每页显示 10 条，查询第 2 页的结果如下：

```
{
  "content": [
    { "id": 11, "name": "User11", "email": "user11@example.com", "age": 25 },
    { "id": 12, "name": "User12", "email": "user12@example.com", "age": 30 },
    ...
  ],
  "totalPages": 10,
  "totalElements": 100,
  "number": 1,
  "size": 10,
  "sort": [
    { "property": "name", "direction": "ASC" }
  ]
}
```

---

4. 总结

- 分页：通过 Pageable 和 PageRequest 实现分页查询。

- 排序：通过 Sort 指定排序字段和方向。
- 分页结果：Page 对象包含分页数据和元信息（如总页数、总记录数）。
- 多字段排序：可以通过 Sort.by() 链式调用实现多字段排序。

Spring Data JPA 的分页和排序功能非常强大且易于使用，适合处理大量数据的分页显示和排序需求。

第一阶段：基础功能实现（内存存储）

项目结构

```
user-management/
├── src/
│   └── main/
│       └── java/io/codescience
│           ├── model/
│           │   └── User.java
│           ├── service/
│           │   └── UserService.java
│           └── ui/
│               └── CliUI.java
└── pom.xml
```

核心代码

01. 领域模型层


```java
// User.java
package io.codescience.model;


public class User {
    private int id;
    private String name;
    private String gender;
    private int age;
    private String email;
    private String phone;

    public User(String name, String gender, int age, String email, String phone) {
        validateInput(name, "姓名");
        validateInput(gender, "性别");
        validateAge(age);
        this.name = name;
        this.gender = gender;
        this.age = age;
        this.email = email;
        this.phone = phone;
    }

    private void validateInput(String value, String field) {
        if (value == null || value.trim().isEmpty()) {
            throw new IllegalArgumentException(field + "不能为空");
        }
    }

    private void validateAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("年龄不能为负数");
        }
    }
    // Getter/Setter methods...
}
```

02. 业务逻辑层

```
// UserService.java
```



```
package io.codescience.service;

import io.codescience.model.User;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class UserService {
    private Map<Integer, User> users = new HashMap<>();
    private int idCounter = 1;

    public User addUser(User user) {
        user.setId(idCounter++);
        users.put(user.getId(), user);
        return user;
    }

    public User deleteUser(int id) {
        return users.remove(id);
    }

    public User updateUser(int id, User newData) {
        User existing = users.get(id);
        if (existing != null) {
            existing.setName(newData.getName());
            existing.setGender(newData.getGender());
            existing.setAge(newData.getAge());
            existing.setEmail(newData.getEmail());
            existing.setPhone(newData.getPhone());
        }
        return existing;
    }

    public User getUser(int id) {
        return users.get(id);
    }

```

```
    public List<User> listUsers() {
        return new ArrayList<>(users.values());
    }
}
```

03. 命令行界面（CLI)


```
// CliUI.java
```

```
import io.codescience.model.User;
import io.codescience.service.UserService;

import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class CliUI {
    private final UserService service = new UserService();

    public CliUI() {
        initSampleData();
    }

    private void initSampleData() {
        // 添加示例用户数据
        User user1 = new User("张三", "男", 25, "zhangsan@example.com", "13800138001");
        User user2 = new User("李四", "女", 28, "lisi@example.com", "13800138002");
        User user3 = new User("王五", "男", 30, "wangwu@example.com", "13800138003");

        service.addUser(user1);
        service.addUser(user2);
        service.addUser(user3);

        System.out.println("示例数据初始化完成");
    }

    public void start() {
        System.out.println("用户管理系统启动（输入help查看命令帮助）");
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.print("\n> ");
            String input = scanner.nextLine().trim();
            if (input.isEmpty())
                continue;

            String[] args = input.split(" ");
```



```
            try {
                switch (args[0]) {
                    case "user":
                        handleUserCommand(args);
                        break;
                    case "exit":
                        System.out.println("系统退出");
                        return;
                    case "help":
                        printHelp();
                        break;
                    default:
                        System.out.println("未知命令");
                }
            } catch (Exception e) {
                System.out.println("错误: " + e.getMessage());
            }
        }
    }

    private void handleUserCommand(String[] args) {
        if (args.length < 2) {
            throw new IllegalArgumentException("命令不完整");
        }

        switch (args[1]) {
            case "add":
                addUserFlow(args);
                break;
            case "delete":
                deleteUserFlow(args);
                break;
            case "list":
                listUsersFlow(args);
                break;
            case "update":
                updateUserFlow(args);
                break;
            default:
                throw new IllegalArgumentException("未知命令: " + args[1]);
```

```
        }
    }

    private void addUserFlow(String[] args) {
        Map<String, String> params = parseParams(args);
        User user = new User(
                params.get("name"),
                params.get("gender"),
                Integer.parseInt(params.get("age")),
                params.get("email"),
                params.get("phone"));
        service.addUser(user);
        System.out.println("用户添加成功，ID: " + user.getId());
    }

    private Map<String, String> parseParams(String[] args) {
        Map<String, String> params = new HashMap<>();
        for (int i = 2; i < args.length; i++) {
            if (args[i].startsWith("--")) {
                String key = args[i].substring(2);
                params.put(key, args[++i]);
            }
        }
        return params;
    }

    private void printHelp() {
        System.out.println("可用命令：");
        System.out.println("user list");
        System.out.println("user add --name <姓名> --gender <性别> --age <年龄> --email <邮箱> --phone <电话>");
        System.out.println("user delete --id <用户ID>");
        System.out.println("user update --id <用户ID> [--name <姓名>] [--gender <性别>] [--age <年龄>] [--email <邮箱>] [--phone <电话>]");
        System.out.println("exit");
    }

    private void deleteUserFlow(String[] args) {
        Map<String, String> params = parseParams(args);
        String userId = params.get("id");
```



```
        if (userId == null) {
```

```
            throw new IllegalArgumentException("缺少用户ID参数");
        }
        try {
            int id = Integer.parseInt(userId);
            service.deleteUser(id);
            System.out.println("用户删除成功");
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("用户ID必须是数字");
        }
    }

    private void listUsersFlow(String[] args) {
        var users = service.listUsers();
        if (users.isEmpty()) {
            System.out.println("当前没有用户");
            return;
        }
        System.out.println("用户列表：");
        for (User user : users) {
            System.out.printf("ID: %d, 姓名: %s, 性别: %s, 年龄: %d, 邮箱: %s, 电话:
%s%n",
                    user.getId(),
                    user.getName(),
                    user.getGender(),
                    user.getAge(),
                    user.getEmail(),
                    user.getPhone());
        }
    }

    private void updateUserFlow(String[] args) {
        Map<String, String> params = parseParams(args);
        String userId = params.get("id");
        if (userId == null) {
            throw new IllegalArgumentException("缺少用户ID参数");
        }

        try {
            int id = Integer.parseInt(userId);
            User user = service.getUser(id);
```



```
            if (user == null) {
```

```
                throw new IllegalArgumentException("用户不存在");
            }

            // 更新用户信息
            if (params.containsKey("name"))
                user.setName(params.get("name"));
            if (params.containsKey("gender"))
                user.setGender(params.get("gender"));
            if (params.containsKey("age"))
                user.setAge(Integer.parseInt(params.get("age")));
            if (params.containsKey("email"))
                user.setEmail(params.get("email"));
            if (params.containsKey("phone"))
                user.setPhone(params.get("phone"));

            service.updateUser(id, user);
            System.out.println("用户信息更新成功");
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("用户ID必须是数字");
    }   }

}
```

04. 程序入口

```
package io.codescience;

im ort io.codescience.ui.CliUI;

pu lic class Application {
    public static void main(String[] args) {
        CliUI cli = new CliUI();
        cli.start();
    }
}
```


运行示例

```
> user add --name "张三" --gender Male --age 30 --email  z@example.com --phone 13800138000
用户添加成功，ID: 1

> user list
1 | 张三 | Male | 30 | z@example.com | 13800138000
```

第二阶段：数据库存储（Repository模式）

新增结构

```
├── repository/
│   ├── UserRepository.java
│   └── UserRepositoryImpl.java
└── config/
    └── DatabaseConfig.java
```

核心代码

01. 数据库配置

```
package io.codescience.config;

public class DatabaseConfig {
    public static final String URL = "jdbc:mysql://localhost:3306/user_management?useSSL=false&serverTimezone=UTC";
    public static final String USER = "root";
    public static final String PASSWORD = "Craftsman@2025";

    static {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("MySQL JDBC Driver not found", e);
        }
    }
}

```

02. 数据访问层

添加POM引用

```
<dependencies>
    <!-- MySQL JDBC Driver -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
</dependencies>
```

UserRepository.java


```
// UserRepository.java
package io.codescience.repository;

import io.codescience.model.User;
import java.util.List;
import java.util.Optional;

public interface UserRepository {
    User save(User user);

    Optional<User> findById(int id);

    List<User> findAll();

    void deleteById(int id);

    void update(User user);
}
```

UserRepositoryImpl

```
package io.codescience.repository;
```



```

import io.codescience.config.DatabaseConfig;
import io.codescience.model.User;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class UserRepositoryImpl implements UserRepository {

    @Override
    public User save(User user) {
        String sql = "INSERT INTO users(name, gender, age, email, phone) VALUES (?,?,?,?,?)";

        try (Connection conn = DriverManager.getConnection(
                DatabaseConfig.URL,
                DatabaseConfig.USER,
                DatabaseConfig.PASSWORD);
                PreparedStatement stmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {

            stmt.setString(1, user.getName());
            stmt.setString(2, user.getGender());
            stmt.setInt(3, user.getAge());
            stmt.setString(4, user.getEmail());
            stmt.setString(5, user.getPhone());

            stmt.executeUpdate();

            try (ResultSet rs = stmt.getGeneratedKeys()) {
                if (rs.next()) {
                    user.setId(rs.getInt(1));
                }
            }
            return user;
        } catch (SQLException e) {
            throw new RuntimeException("保存用户失败", e);
        }
```

```
    }

    @Override
    public Optional<User> findById(int id) {
        String sql = "SELECT * FROM users WHERE id = ?";

        try (Connection conn = DriverManager.getConnection(
                DatabaseConfig.URL,
                DatabaseConfig.USER,
                DatabaseConfig.PASSWORD);
                PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setInt(1, id);

            try (ResultSet rs = stmt.executeQuery()) {
                if (rs.next()) {
                    return Optional.of(mapResultSetToUser(rs));
                }
            }
            return Optional.empty();
        } catch (SQLException e) {
            throw new RuntimeException("查询用户失败", e);
        }
    }

    @Override
    public List<User> findAll() {
        String sql = "SELECT * FROM users";
        List<User> users = new ArrayList<>();

        try (Connection conn = DriverManager.getConnection(
                DatabaseConfig.URL,
                DatabaseConfig.USER,
                DatabaseConfig.PASSWORD);
                PreparedStatement stmt = conn.prepareStatement(sql);
                ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {
                users.add(mapResultSetToUser(rs));
            }
```



```
            return users;
        } catch (SQLException e) {
            throw new RuntimeException("查询用户列表失败", e);
        }
    }

    @Override
    public void deleteById(int id) {
        String sql = "DELETE FROM users WHERE id = ?";

        try (Connection conn = DriverManager.getConnection(
                DatabaseConfig.URL,
                DatabaseConfig.USER,
                DatabaseConfig.PASSWORD);
                PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setInt(1, id);
            stmt.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException("删除用户失败", e);
        }
    }

    @Override
    public void update(User user) {
        String sql = "UPDATE users SET name = ?, gender = ?, age = ?, email = ?, phone = ? WHERE id = ?";

        try (Connection conn = DriverManager.getConnection(
                DatabaseConfig.URL,
                DatabaseConfig.USER,
                DatabaseConfig.PASSWORD);
                PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setString(1, user.getName());
            stmt.setString(2, user.getGender());
            stmt.setInt(3, user.getAge());
            stmt.setString(4, user.getEmail());
            stmt.setString(5, user.getPhone());
            stmt.setInt(6, user.getId());
```

```
118
```



```
119             int affectedRows = stmt.executeUpdate();
120             if (affectedRows == 0) {
121                 throw new IllegalArgumentException("用户不存在");
122             }
123         } catch (SQLException e) {
124             throw new RuntimeException("更新用户失败", e);
125         }
126     }
127 
128     private User mapResultSetToUser(ResultSet rs) throws SQLException {
129         User user = new User();
130         user.setId(rs.getInt("id"));
131         user.setName(rs.getString("name"));
132         user.setGender(rs.getString("gender"));
133         user.setAge(rs.getInt("age"));
134         user.setEmail(rs.getString("email"));
135         user.setPhone(rs.getString("phone"));
136         return user;
137     }
138 }
```

服务层改造

```
// UserService.java
```



```
package io.codescience.service;

import io.codescience.model.User;
import io.codescience.repository.UserRepository;
import io.codescience.repository.UserRepositoryImpl;

import java.util.List;

public class UserService {
    private final UserRepository repository;

    public UserService() {
        this.repository = new UserRepositoryImpl();
    }

    public User addUser(User user) {
        return repository.save(user);
    }

    public User getUser(int id) {
        return repository.findById(id)
                .orElseThrow(() -> new IllegalArgumentException("用户不存在"));
    }

    public List<User> listUsers() {
        return repository.findAll();
    }

    public void deleteUser(int id) {
        if (!repository.findById(id).isPresent()) {
            throw new IllegalArgumentException("用户不存在");
        }
        repository.deleteById(id);
    }

    public void updateUser(int id, User user) {
        if (!repository.findById(id).isPresent()) {
            throw new IllegalArgumentException("用户不存在");
```

```
        }
        user.setId(id);
        repository.update(user);
    }
}
```

数据库初始化脚本

#04. Docker 数据管理（Volumes） （可选）

```
CREATE DATABASE IF NOT EXISTS user_management;
USE user_management;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    gender ENUM('Male','Female','Other') NOT NULL,
    age INT,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(15)
);
```

第三阶段：换底层数据库访问逻辑

一、MyBatis Repository的实现

1. 添加依赖

首先在 `pom.xml` 中添加必要的依赖：

```
<dependencies>
    <!-- MySQL JDBC Driver -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>

    <!-- MyBatis -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.13</version>
    </dependency>
</dependencies>

```

2. 创建数据库配置

创建 `DatabaseConfig.java` ：

```
package io.codescience.config;

public class DatabaseConfig {
    public static final String URL = "jdbc:mysql://localhost:3306/user_management?useSSL=false&serverTimezone=UTC";
    public static final String USER = "root";
    public static final String PASSWORD = "your_password";

    static {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("MySQL JDBC Driver not found", e);
        }
    }
}
```

```
16
```

3. 创建 MyBatis 配置文件

创建 `src/main/resources/mybatis-config.xml` ：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/user_management?
useSSL=false&amp;serverTimezone=UTC"/>
                <property name="username" value="root"/>
                <property name="password" value="your_password"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="mapper/UserMapper.xml"/>
    </mappers>
</configuration>

```

4. 创建 Mapper 接口

创建 `UserMapper.java` ：

```java
package io.codescience.repository;

import io.codescience.model.User;
import java.util.List;
import java.util.Optional;

public interface UserMapper {
    void insert(User user);
    Optional<User> findById(int id);
    List<User> findAll();
    void deleteById(int id);
    void update(User user);
}

```

5. 创建 Mapper XML

创建 `src/main/resources/mapper/UserMapper.xml` ：


```xml
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="io.codescience.repository.UserMapper">
    <resultMap id="userMap" type="io.codescience.model.User">
        <id property="id" column="id"/>
        <result property="name" column="name"/>
        <result property="gender" column="gender"/>
        <result property="age" column="age"/>
        <result property="email" column="email"/>
        <result property="phone" column="phone"/>
    </resultMap>

    <insert id="insert" parameterType="io.codescience.model.User"
useGeneratedKeys="true" keyProperty="id">
        INSERT INTO users (name, gender, age, email, phone)
        VALUES (#{name}, #{gender}, #{age}, #{email}, #{phone})
    </insert>

    <select id="findById" resultMap="userMap">
        SELECT * FROM users WHERE id = #{id}
    </select>

    <select id="findAll" resultMap="userMap">
        SELECT * FROM users
    </select>

    <delete id="deleteById">
        DELETE FROM users WHERE id = #{id}
    </delete>

    <update id="update" parameterType="io.codescience.model.User">
        UPDATE users
        SET name = #{name},
            gender = #{gender},
            age = #{age},
            email = #{email},
            phone = #{phone}
        WHERE id = #{id}
    </update>
</mapper>
```


6. 实现 Repository

创建 `MyBatisUserRepository.java` ：

```
package io.codescience.repository;
```



```

import io.codescience.model.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;
import java.util.Optional;

public class MyBatisUserRepository implements UserRepository {
    private final SqlSessionFactory sqlSessionFactory;

    public MyBatisUserRepository() {
        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            throw new RuntimeException("初始化 MyBatis 失败", e);
        }
    }

    @Override
    public User save(User user) {
        try (SqlSession session = sqlSessionFactory.openSession()) {
            UserMapper mapper = session.getMapper(UserMapper.class);
            mapper.insert(user);
            session.commit();
            return mapper.findById(user.getId())
                    .orElseThrow(() -> new RuntimeException("保存用户后无法获取用户信
息"));
        }
    }

    @Override
    public Optional<User> findById(int id) {
```

```
        try (SqlSession session = sqlSessionFactory.openSession()) {
            UserMapper mapper = session.getMapper(UserMapper.class);
            return mapper.findById(id);
        }
    }

    @Override
    public List<User> findAll() {
        try (SqlSession session = sqlSessionFactory.openSession()) {
            UserMapper mapper = session.getMapper(UserMapper.class);
            return mapper.findAll();
        }
    }

    @Override
    public void deleteById(int id) {
        try (SqlSession session = sqlSessionFactory.openSession()) {
            UserMapper mapper = session.getMapper(UserMapper.class);
            mapper.deleteById(id);
            session.commit();
        }
    }

    @Override
    public void update(User user) {
        try (SqlSession session = sqlSessionFactory.openSession()) {
            UserMapper mapper = session.getMapper(UserMapper.class);
            mapper.update(user);
            session.commit();
        }
    }
}
```

7. 修改 Service 层

修改 `UserService.java` ：


```
public class UserService {
    private final UserRepository repository;

    public UserService() {
        this.repository = new MyBatisUserRepository();
    }
    // ... 其他方法保持不变
}
```

重要说明：

1. **配置说明**：
   - 确保数据库连接信息正确（URL、用户名、密码）
   - 数据库和表需要提前创建
   - MyBatis 配置文件必须放在 resources 目录下
2. **事务管理**：
   - 每个方法都使用独立的 SqlSession
   - 写操作（insert、update、delete）后需要调用 commit()
   - 使用 try-with-resources 自动关闭 SqlSession
3. **错误处理**：
   - 所有数据库操作都包含在 try-catch 块中
   - 适当的错误信息转换
   - 事务回滚处理
4. **性能考虑**：
   - SqlSessionFactory 是单例的
   - 使用连接池（POOLED）
   - 及时关闭资源
5. **使用建议**：
   - 在开发环境中使用
   - 生产环境建议使用连接池（如 HikariCP）
   - 考虑添加日志记录
   - 可以添加缓存机制

测试步骤：

1. 确保 MySQL 服务运行
2. 创建数据库和表

3. 运行程序
4. 测试基本操作：

```
user add --name Jessica --gender Female --age 27 --email 1111 --phone 1111111
user list
user update --id 1 --name Jessica2
user delete --id 1

```

这个实现提供了：

- 清晰的代码结构
- 良好的错误处理
- 事务管理
- 资源管理
- 可扩展性

二、Hibernate Repository的实现

1. 环境准备

1.1 添加依赖

在 `pom.xml` 中添加以下依赖：


```
<dependencies>
    <!-- Hibernate Core -->
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.1.Final</version>
    </dependency>

    <!-- MySQL Connector -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>8.3.0</version>
    </dependency>
</dependencies>

```

1.2 创建配置文件

在 `src/main/resources` 目录下创建 `hibernate.cfg.xml` ：

```xml
<?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>
        <!-- 数据库连接配置 -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/user_management?useSSL=false&amp;serverTimezone=UTC</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>

        <!-- Hibernate 配置 -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- 实体类映射 -->
        <mapping class="io.codescience.model.User"/>
    </session-factory>
</hibernate-configuration>

```

2. 实现步骤

2.1 创建实现类

创建 `HibernateUserRepositoryImpl` 类，实现 `UserRepository` 接口：


```java
package io.codescience.repository;

import io.codescience.model.User;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.Query;

import java.util.List;
import java.util.Optional;

public class HibernateUserRepositoryImpl implements UserRepository {
    private final SessionFactory sessionFactory;

    public HibernateUserRepositoryImpl() {
        try {
            Configuration configuration = new Configuration();
            configuration.configure("hibernate.cfg.xml");
            sessionFactory = configuration.buildSessionFactory();
        } catch (Exception e) {
            throw new RuntimeException("初始化 Hibernate 失败", e);
        }
    }
}

```

2.2 实现 save 方法

```
@Override
public User save(User user) {
    try (Session session = sessionFactory.openSession()) {
        Transaction transaction = session.beginTransaction();
        try {
            session.persist(user);
            transaction.commit();
            return user;
        } catch (Exception e) {
            transaction.rollback();
            throw e;
        }
    }
}

```

2.3 实现 findById 方法

```
@Override
public Optional<User> findById(int id) {
    try (Session session = sessionFactory.openSession()) {
        User user = session.get(User.class, id);
        return Optional.ofNullable(user);
    }
}

```

2.4 实现 findAll 方法


```
@Override
public List<User> findAll() {
    try (Session session = sessionFactory.openSession()) {
        Query<User> query = session.createQuery("FROM User", User.class);
        return query.list();
    }
}

```

2.5 实现 deleteById 方法

```
@Override
public void deleteById(int id) {
    try (Session session = sessionFactory.openSession()) {
        Transaction transaction = session.beginTransaction();
        try {
            User user = session.get(User.class, id);
            if (user != null) {
                session.remove(user);
            }
            transaction.commit();
        } catch (Exception e) {
            transaction.rollback();
            throw e;
        }
    }
}

```

2.6 实现 update 方法

```
@Override
public void update(User user) {
    try (Session session = sessionFactory.openSession()) {
        Transaction transaction = session.beginTransaction();
        try {
            session.merge(user);
            transaction.commit();
        } catch (Exception e) {
            transaction.rollback();
            throw e;
        }
    }
}

```

3. 关键点说明

3.1 Session 管理

- 使用 try-with-resources 自动管理 Session 的生命周期
- 确保 Session 在使用后正确关闭
- 避免 Session 泄漏

3.2 事务管理

- 每个写操作（save、delete、update）都需要事务
- 使用 try-catch 处理事务异常
- 发生异常时回滚事务

3.3 异常处理

- 初始化异常：包装为 RuntimeException
- 数据库操作异常：在事务中处理
- 查询异常：返回 Optional 或空列表

3.4 性能优化

- 使用 Session 的 get 方法进行主键查询
- 使用 HQL 进行复杂查询
- 合理使用事务范围

4. 使用示例

4.1 创建用户


```
User user = new User();
user.setName("张三");
user.setEmail("zhangsan@example.com");
userRepository.save(user);

```

4.2 查询用户

```
Optional<User> user = userRepository.findById(1);
user.ifPresent(u -> System.out.println(u.getName()));

```

4.3 更新用户

```
User user = userRepository.findById(1).orElseThrow();
user.setName("李四");
userRepository.update(user);

```

4.4 删除用户

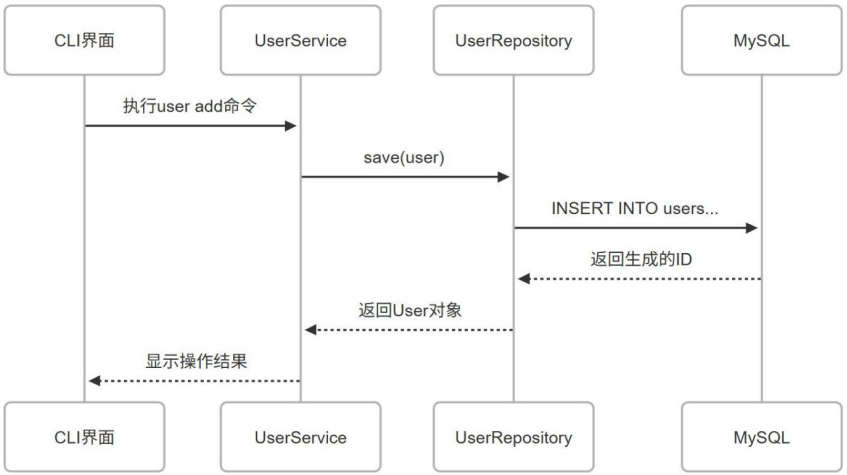
```
userRepository.deleteById(1);
```

分层架构设计（仅仅是介绍）

完整项目结构

```
src/
└── main/
    ├── java/
    │   ├── model/User.java
    │   ├── repository/UserRepository.java
    │   ├── service/UserService.java
    │   ├── ui/CliUI.java
    │   └── exception/
    │       ├── DataAccessException.java
    │       └── ValidationException.java
    └── resources/
        └── db/
            └── migration/
                └── V1__Create_users_table.sql
```

分层调用流程




```
sequenceDiagram
    participant UI as CLI界面
    participant Service as UserService
    participant Repository as UserRepository
    participant DB as MySQL

    UI->>Service: 执行user add命令
    Service->>Repository: save(user)
    Repository->>DB: INSERT INTO users...
    DB-->>Repository: 返回生成的ID
    Repository-->>Service: 返回User对象
    Service-->>UI: 显示操作结果
```

关键设计原则

1. **依赖倒置**

```
// Service层通过接口依赖Repository
public interface UserRepository {
    User save(User user);
    User findById(int id);
    // 其他方法...
}

// 实现类
public class JdbcUserRepository implements UserRepository {
    // JDBC具体实现...
}
```

1. **异常处理分层**

```java
// 自定义异常类
public class DataAccessException extends RuntimeException {
    public DataAccessException(String message, Throwable cause) {
        super(message, cause);
    }
}

// Service层抛出业务异常
public class ValidationException extends RuntimeException {
    public ValidationException(String message) {
        super(message);
    }
}
```

1. 输入验证增强

```java
public User addUser(User user) {
    validateEmailFormat(user.getEmail());
    validatePhoneFormat(user.getPhone());
    return repository.save(user);
}

private void validateEmailFormat(String email) {
    if (!email.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
        throw new ValidationException("邮箱格式不正确");
    }
}
```

扩展功能建议

1. 日志记录


```java
// 添加日志依赖
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.36</version>
</dependency>

// 在关键位置添加日志
private static final Logger logger = LoggerFactory.getLogger(UserService.class);

public User addUser(User user) {
    logger.info("添加用户: {}", user.getName());
    // ...
}
```

1. 分页查询优化

```java
public List<User> listUsers(int page, int pageSize) {
    String sql = "SELECT * FROM users LIMIT ? OFFSET ?";
    int offset = (page - 1) * pageSize;
    // 执行分页查询...
}
```

1. 连接池配置

```java
// 使用HikariCP连接池
HikariConfig config = new HikariConfig();
config.setJdbcUrl(DatabaseConfig.URL);
config.setUsername(DatabaseConfig.USER);
config.setPassword(DatabaseConfig.PASSWORD);
HikariDataSource ds = new HikariDataSource(config);
```

用户管理系统

用户管理系统业务需求

1. 概述

本系统旨在通过命令行界面（CLI）实现一个简单的用户管理系统。用户可以通过命令行进行增、删、改、查操作，并且系统将用户信息持久化到数据库中。系统将采用分层架构设计，确保代码的可维护性和扩展性。

2. 功能需求

2.1 用户管理功能

- 增加用户：用户可以通过命令行输入用户信息（姓名、性别、年龄、电子邮件、电话号码），系统将生成一个唯一的用户ID，并将用户信息存储到数据库中。
- 删除用户：用户可以通过输入用户ID删除指定的用户信息。
- 修改用户：用户可以通过输入用户ID修改指定用户的姓名、性别、年龄、电子邮件、电话号码等信息。
- 查询用户：用户可以通过输入用户ID查询指定用户的详细信息，或者列出所有用户的信息。

2.2 命令行界面（CLI）

- 操作菜单：系统启动后，展示一个操作菜单，用户可以选择进行增、删、改、查操作。
- 命令格式：系统支持类似 git 或 Docker 的命令格式，用户可以通过输入命令和参数来执行相应的操作。
  - 示例命令：
    - user add --name "John Doe" --gender Male --age 30 --email john@example.com --phone 1234567890
    - user delete --id 1
    - user update --id 1 --name "Jane Doe" --email jane@example.com
    - user get --id 1
    - user list

2.3 数据库存储

- 数据库：使用 MySQL 作为关系型数据库。
- 表结构：在 user_management 数据库中创建 users 表，表结构如下：


```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    gender ENUM('Male', 'Female', 'Other') NOT NULL,
    age INT,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(15)
);
```

3. 非功能需求

3.1 性能

- 系统应能够快速响应用户的操作请求，尤其是在查询和列出用户信息时，应尽量减少数据库查询时间。

3.2 安全性

- 系统应对用户输入进行验证，防止 SQL 注入等安全问题。
- 数据库连接信息应妥善保管，避免泄露。

3.3 可维护性

- 代码应具有良好的结构和注释，便于后续维护和扩展。
- 系统应采用模块化设计，确保各层之间的职责清晰。

3.4 可扩展性

- 系统应设计为易于扩展，未来可以方便地添加新的功能模块或修改现有功能。

4. 分层设计与架构

1. 分层架构：
   - 将系统设计为三层架构，包括：
     - 表示层（UI层）： 负责与用户交互，通过命令行界面显示操作菜单，并接收用户输入。
     - 业务逻辑层（Service层）： 负责处理业务逻辑，如验证输入、执行增、删、改、查操作。
     - 数据访问层（DAO层）： 负责与数据库交互，提供增、删、改、查的数据库操作方法。
2. Repository与分层结合：
   - 在 Service 层调用 Repository 类进行数据存储操作，将数据访问逻辑与业务逻辑分开，提高系统的可维护性和扩展性。
3. 代码组织：
   - com.example.model：定义用户模型类（User），包括用户属性。
   - com.example.dao：实现数据库操作类（UserRepository），使用 JDBC 进行数据存取。
   - com.example.service：实现业务逻辑层，调用 Repository 层进行增、删、改、查操作。
   - com.example.ui：提供命令行交互界面，展示操作菜单，接收用户输入。

5. 测试需求（可选-根据能力实现）

- 单元测试：对 UserRepository 和 UserService 进行单元测试，确保每个方法的功能正确。
- 集成测试：测试整个系统的功能，确保从命令行输入到数据库操作的整个流程正确无误。
- 性能测试：测试系统在高并发情况下的性能表现，确保系统能够处理大量用户请求。

6. 未来扩展

- 用户权限管理：未来可以扩展系统，增加用户权限管理功能，不同权限的用户可以执行不同的操作。
- 日志记录：增加日志记录功能，记录用户的操作历史，便于审计和排查问题。
- 多数据库支持：未来可以扩展系统，支持多种数据库（如 PostgreSQL、Oracle 等）。

第一阶段：实现基本功能

1. 功能要求：
    - 实现用户的增、删、改、查操作。
    - 每个用户应包含以下信息：
        - 用户ID（唯一标识符）
        - 姓名
        - 性别
        - 年龄
        - 电子邮件
        - 电话号码
2. 命令行UI：
    - 在命令行界面展示一个操作菜单，用户可选择进行增、删、改、查操作。
    - 提供用户输入信息并执行相应的操作。

第二阶段：数据库存储与Repository模式

1. 数据库要求：
    - 使用 MySQL 或其他关系型数据库。
    - 创建一个 user_management 数据库，并在其中创建一个 users 表，字段与用户信息相匹配。
2. Repository模式：
    - 实现一个 Repository 类，负责与数据库交互，提供增、删、改、查等操作。
    - 通过 JDBC 连接数据库，并确保数据存储与读取正确。
3. 功能增强：

    - 将用户信息持久化到数据库中，确保增、删、改、查操作都能反映在数据库中。

第三阶段：更换底层数据库访问逻辑

目前使用JDBC，请尝试使用MyBatis和 Hibernate 实现Repository（此部分没有提供示例 - 扩展功能）。