第4章 Spring的数据库编程

湖南科技大学
计算机科学与工程学院

学习目标/Target

- 了解JdbcTemplate类的作用
- 熟悉Spring JDBC的配置
- 熟悉JdbcTemplate的增删改查操作

学习目标/Target

- 理解Spring事务管理
- 掌握基于XML方式的声明式事务
- 熟悉基于注解方式的声明式事务

章节概述/ Summary

数据库用于处理持久化业务产生的数据，应用程序在运行过程中经常要操作数据库。一般情况下，数据库的操作由持久层来实现。作为扩展性较强的一站式开发框架，Spring也提供了持久层Spring JDBC功能，Spring JDBC可以管理数据库连接资源，简化传统JDBC的操作，进而提升程序数据库操作的效率。本章将对Spring JDBC相关知识进行详细讲解。

目录/Contents

4.1 Spring JDBC

4.1.1 JDBCTemplate概述

- 了解JDBCTemplate概述，能够说出JDBCTemplate的作用

4.1.1 JDBCTemplate概述

JDBCTemplate作用

针对数据库操作，Spring框架提供了JdbcTemplate类，JdbcTemplate是一个模板类，Spring JDBC中的更高层次的抽象类均在JdbcTemplate模板类的基础上创建。JdbcTemplate类提供了操作数据库的基本方法，包括添加、删除、查询和更新。在操作数据库时，JdbcTemplate类简化了传统JDBC中的复杂步骤，这可以让开发人员将更多精力投入到业务逻辑中。

4.1.1 JDBCTemplate概述

抽象类JdbcAccessor的属性

JdbcTemplate类继承自抽象类JdbcAccessor，同时实现了JdbcTemplate接口。抽象类JdbcAccessor提供了一些访问数据库时使用的公共属性，具体如下：

- DataSource：DataSource主要功能是获取数据库连接。在具体的数据操作中，它还提供对数据库连接的缓冲池和分布式事务的支持。
- SQLExceptionTranslator：SQLExceptionTranslator是一个接口，它负责对SQLException异常进行转译工作。

4.1.2 Spring JDBC的配置

- 熟悉Spring JDBC的配置，能够在XML文件中完成Spring JDBC的配置

4.1.2 Spring JDBC的配置

Spring JDBC中的4个包说明

包名	说明
core（核心包）	包含了JDBC的核心功能,包括JdbcTemplate类、SimpleJdbcInsert类、SimpleJdbcCall类以及NamedParameterJdbcTemplate类。
dataSource（数据源包）	包含访问数据源的实用工具类,它有多种数据源的实现,可以在Java EE容器外部测试JDBC代码。
object（对象包）	以面向对象的方式访问数据库,它可以执行查询、修改和更新操作并将返回结果作为业务对象，并且在数据表的列和业务对象的属性之间映射查询结果。
support（支持包）	包含了core和object包的支持类，如提供异常转换功能的SQLException类。

4.1.2 Spring JDBC的配置

Spring对数据库的操作都封装在了core、dataSource、object和support这4个包中，想要使用Spring JDBC，就需要对这些包进行配置。

在Spring中，JDBC的配置是在配置文件applicationContext.xml中完成的，包括配置数据源、配置JDBC模板和配置注入类。

4.1.2 Spring JDBC的配置

```
<!-- 1.配置数据源 -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <!-- 数据库驱动 -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <!-- 连接数据库url -->
    <property name="url" value="jdbc:mysql://localhost:3306/spring"/>
    <property name="username" value="root"/><!-- 连接数据库用户名 -->
    <property name="password" value="root"/><!-- 连接数据库密码 -->
</bean>
<!-- 2.配置JDBC模板 -->
<bean id="JdbcTemplate"   class="org.springframework.jdbc.core.JdbcTemplate">
      <property name="dataSource" ref="dataSource"/>
</bean>
<!-- 3.配置注入类 -->
<bean id="xxx" class="Xxx">  <property name="JdbcTemplate" ref="JdbcTemplate"/></bean>
```

4.1.2 Spring JDBC的配置

dataSource配置4个属性的含义

属性名	含义
driverClassName	所使用的驱动名称,对应驱动JAR包中的Driver类
url	数据源地址
username	访问数据库的用户名
password	访问数据库的密码

4.1.2 Spring JDBC的配置

dataSource属性值的设定要求

在dataSource的4个属性中，需要根据数据库类型或者系统配置设置相应的属性值。例如，如果数据库类型不同，需要更改驱动名称；如果数据库不在本地，则需要将地址中的localhost替换成相应的主机IP；默认情况下，数据库端口号可以省略，但如果修改过MySQL数据库的端口号，则需要加上修改后的端口号。此外，连接数据库的用户名和密码需要与数据库创建时设置的用户名和密码保持一致。

4.2 JDBCTemplate的常用方法

4.2.1 excute()方法

- 熟悉excute()方法，能够在程序中熟练使用excute()方法执行SQL语句

4.2.1 excute()方法

execute()方法用于执行SQL语句，其语法格式如下：

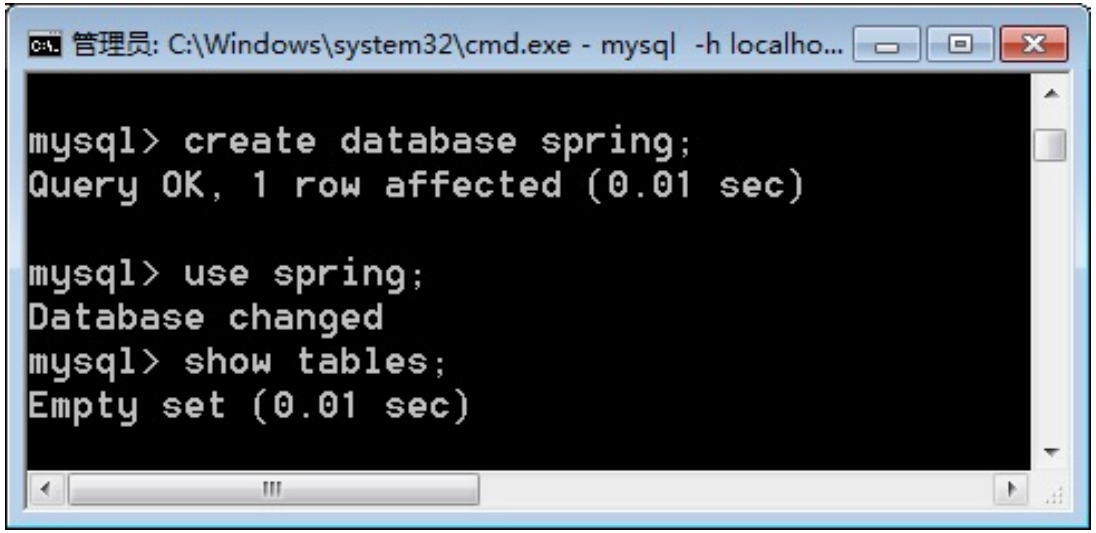
```
jdTemplate.execute("SQL 语句");
```

4.2.1 excute()方法

下面以创建数据表的SQL语句为例，来演示excute()方法的使用，具体步骤如下。

STEP 01

创建数据库：在MySQL中，创建一个名为spring的数据库。



4.2.1 excute()方法

STEP 02

创建项目并引入依赖：在IDEA中创建一个名为chapter09的Maven项目，然后在pom.xml文件中加载使用到的 **Spring 基础包**、**Spring依赖包**、**MySQL数据库的驱动JAR包**、**Spring JDBC的JAR包**和**Spring事务处理的JAR包**。

```
<!– 这里只展示了其中一个JAR包-->
<!-- MySQL数据库驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
        <scope>runtime</scope>
    </dependency>
```

4.2.1 excute()方法

STEP 03

编写配置文件：在chapter09项目的src/main/resources目录下，创建配置文件applicationContext.xml，在该文件中配置数据源Bean和JDBC模板Bean，并将数据源注入到JDBC模板中。

```
<!-- 1. 配置数据源 -->
<bean id="dataSource" class=
"org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost/spring" />
        <property name="username" value="root" />
        <property name="password" value="root" /></bean>
<!-- 2. 配置 JDBC 模板 -->
<bean id="jdbcTemplate"      class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" /></bean>
```

4.2.1 excute()方法

STEP 04

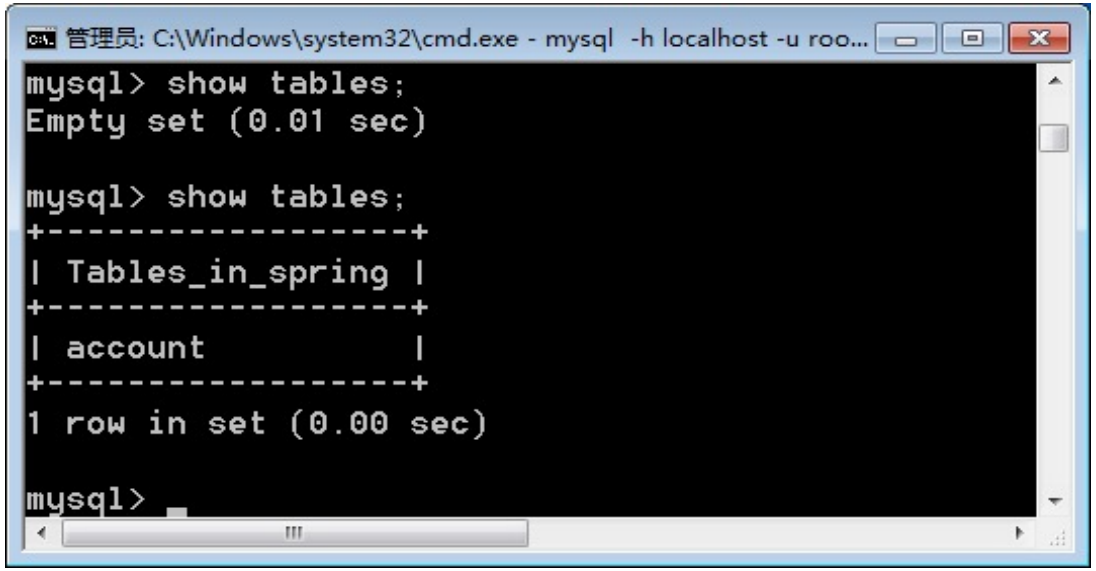
编写测试类：创建测试类TestJdbcTemplate，在该类的main()方法中获取JdbcTemplate实例，然后调用execute()方法执行创建数据表的SQL语句。

```
public class TestJdbcTemplate {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
            ClassPathXmlApplicationContext("applicationContext.xml");
        JdbcTemplate jdTemplate =
                (JdbcTemplate) applicationContext.getBean("jdbcTemplate");
        jdTemplate.execute("create table account(" +
                "id int primary key auto_increment," +
                 “username varchar(50)," +              "balance double)");
        System.out.println("账户表account创建成功！");}
}
```

4.2.1 excute()方法

STEP 05

查看运行结果：在IDEA中启动TestJdbcTemplate类，再次查询spring数据库。



4.2.2 update()方法

- 熟悉update()方法，能够在程序中使用update()方法进行数据的增删改操作

4.2.2 update()方法

JdbcTemplate类中常用的update()方法

方法	说明
int update(String sql)	该方法是最简单的update()方法重载形式,它直接执行传入的SQL语句,并返回受影响的行数。
int update(PreparedStatementCreator psc)	该方法执行参数psc返回的语句,然后返回受影响的行数。
int update(String sql, PreparedStatementSetter pss)	该方法通过参数pss设置SQL语句中的参数,并返回受影响的行数。
int update(String sql,Object... args)	该方法可以为SQL语句设置多个参数,这些参数保存在参数args中,使用Object...设置SQL语句中的参数,要求参数不能为NULL,并返回受影响的行数。

4.2.2 update()方法

下面通过一个案例演示update()方法的使用，该案例要求添加、更新、删除用户账户。案例具体实现步骤如下所示。

STEP 01

编写实体类：创建Account类，在该类中定义属性，以及其对应的getter/setter方法。

```
public class Account {
    private Integer id;                    // 账户id
    private String username;               // 用户名
    private Double balance;                // 账户余额
    // 省略gettter/setter方法
    public String toString() {
        return "Account [id=" + id + ", "
        + "username=" + username + ", balance=" + balance + "]";}
}
```

4.2.2 update()方法

STEP 02

编写Dao层接口：创建接口AccountDao，并在接口中定义添加、更新和删除账户的方法。

```
public interface AccountDao {
        // 添加
        public int addAccount(Account account);
        // 更新
        public int updateAccount(Account account);
        // 删除
        public int deleteAccount(int id);
}
```

4.2.2 update()方法

STEP 03

实现Dao层接口：创建AccountDao接口的实现类AccountDaoImpl，并在类中实现添加、更新和删除账户的方法。

```
public class AccountDaoImpl implements AccountDao {
    // 定义JdbcTemplate属性，此处省略setter方法
    private JdbcTemplate jdbcTemplate;
    // 这里只展示（添加账户）的操作
    public int addAccount(Account account) {
        String sql = "insert into account(username,balance) value(?,?)";
        Object[] obj = new Object[] {// 定义数组来存放SQL语句中的参数
            account.getUsername(), account.getBalance()          };
          // 执行添加操作，返回的是受SQL语句影响的记录条数
        return this.jdbcTemplate.update(sql, obj);}
}
```

4.2.2 update()方法

STEP 04

编写配置文件：在applicationContext.xml中，定义一个id为accountDao的Bean，用于将jdbcTemplate注入到accountDao实例中。

```
<!--定义id为accountDao的Bean-->
<bean id="accountDao"
class="com.itheima.AccountDaoImpl">
        <!-- 将jdbcTemplate注入到accountDao实例中 -->
        <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

4.2.2 update()方法

STEP 05

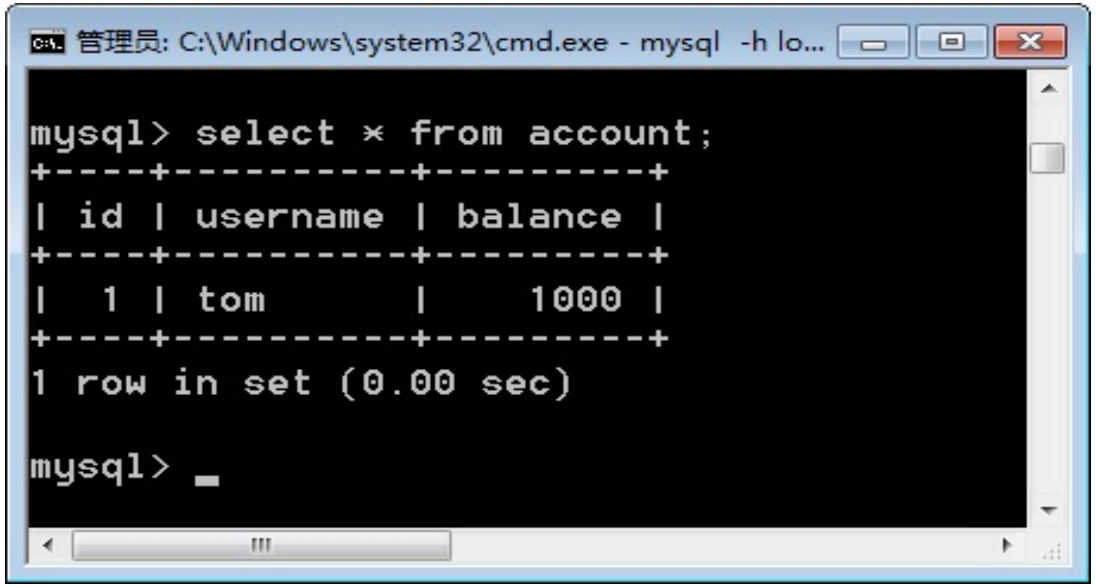
测试添加功能：创建测试类TestAddAccount，该类主要用于添加用户账户信息。

```
public class TestAddAcount {
    public static void main(String[] args) {
        ApplicationContext applicationContext =new
            ClassPathXmlApplicationContext("applicationContext.xml");
        AccountDao accountDao =
                (AccountDao) applicationContext.getBean("accountDao");
        Account account = new Account();
        account.setUsername("tom");     account.setBalance(1000.00);
        int num = accountDao.addAccount(account);
        if (num > 0) { System.out.println("成功插入了" + num + "条数据！");
        } else { System.out.println("插入操作执行失败！"); }
}}
```

4.2.2 update()方法

STEP 06

查看执行第5步后的运行结果：在IDEA中启动TestAddAccount类，控制台会输出结果。此时再次查询spring数据库中的account表。



4.2.2 update()方法

STEP 07

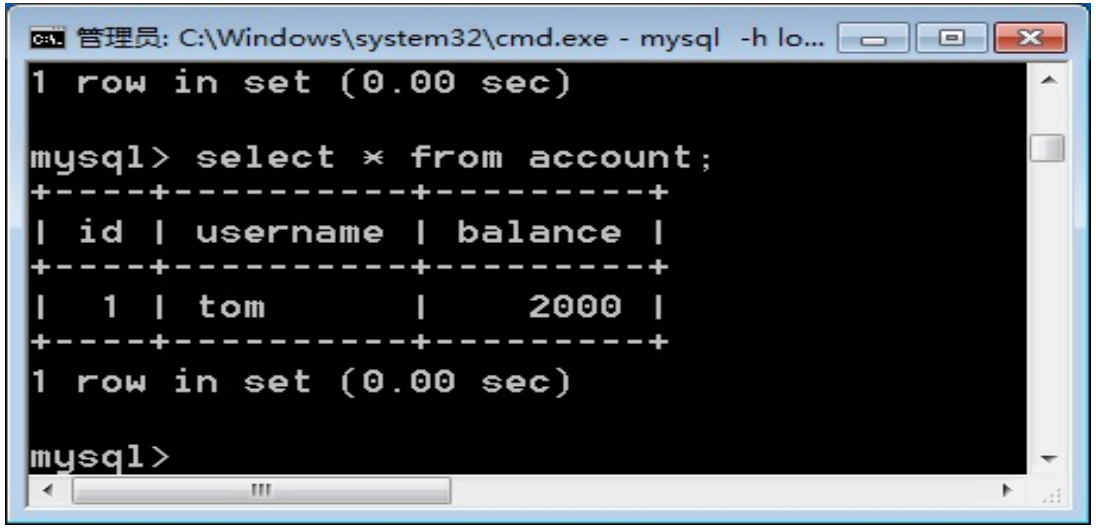
测试更新功能：执行完插入操作后，接下来调用JdbcTemplate类的update()方法执行更新操作。

```
public class TestUpdateAccount {
    public static void main(String[] args) {
        ApplicationContext applicationContext =new
            ClassPathXmlApplicationContext("applicationContext.xml");
        AccountDao accountDao =
                (AccountDao) applicationContext.getBean("accountDao");
        Account account = new Account();
        account.setId(1);          account.setUsername("tom");
        account.setBalance(2000.00);
        int num = accountDao.updateAccount(account);
        if (num > 0) {System.out.println("成功修改了" + num + "条数据！");
        } else {System.out.println("修改操作执行失败！");}     }}
```

4.2.2 update()方法

STEP 08

查看执行第7步后的运行结果：在IDEA中启动TestUpdateAccount类，控制台会输出结果。此时再次查询spring数据库中的account表。



4.2.2 update()方法

STEP 09

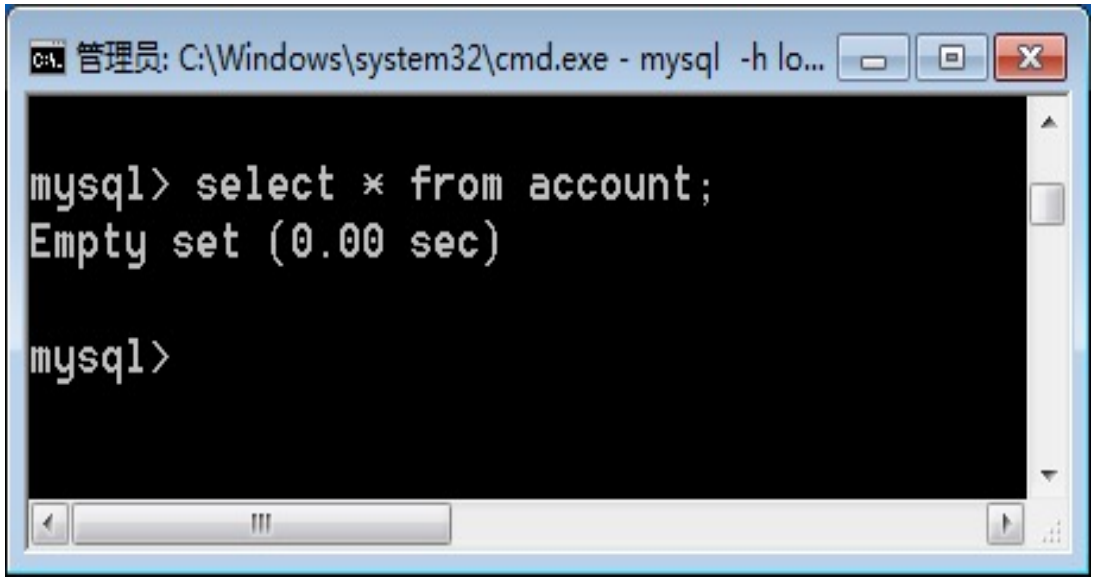
测试删除功能：创建测试类TestDeleteAccount，该类主要用于测试删除用户账户信息。

```
public class TestDeleteAccount {
    public static void main(String[] args) {
        // 加载配置文件
        ApplicationContext applicationContext =new
            ClassPathXmlApplicationContext("applicationContext.xml");
        // 获取AccountDao实例
        AccountDao accountDao =
                (AccountDao) applicationContext.getBean("accountDao");
        // 执行deleteAccount()方法，并获取返回结果
        int num = accountDao.deleteAccount(1);
        // 输出语句省略
}}
```

4.2.2 update()方法

STEP 10

查看执行第9步后的运行结果：在IDEA中启动TestDeleteAccount类，控制台会输出结果。此时再次查询spring数据库中的account表。



4.2.3 query()方法

- 熟悉query()方法，能够在程序中使用query()方法进行数据查询操作

4.2.3 query()方法

JdbcTemplate类中常用的查询方法

方法	说明
List query(String sql, RowMapper rowMapper)	执行String类型参数提供的SQL语句,并通过参数rowMapper返回一个List类型的结果。
List query(String sql, PreparedStatementSetter pss, RowMapper rowMapper)	根据String类型参数提供的SQL语句创建PreparedStatement对象,通过参数rowMapper将结果返回到List中。
List query(String sql,Object[] args, RowMapper rowMapper)	使用Object[]的值来设置SQL语句中的参数值,rowMapper是个回调方法,直接返回List类型的数据。
queryForObject(String sql,RowMapper rowMapper,Object... args)	将args参数绑定到SQL语句中,并通过参数rowMapper返回一个Object类型的单行记录。
queryForList(String sql,Object[] args, class<T> elementType)	该方法可以返回多行数据的结果,但必须返回列表,args参数是sql语句中的参数,elementType参数返回的是List数据类型。

4.2.3 query()方法

了解了JdbcTemplate类中几个常用的query()方法后，接下来通过一个具体的案例演示query()方法的使用，案例实现步骤如下。

STEP 01

插入数据：向数据表account中插入几条数据。

```
insert into `account`(`id`,`username`,`balance`)
values
 (1,'zhangsan',100),(3,'lisi',500),(4,'wangwu',300);
```

4.2.3 query()方法

STEP 02

编写查询方法：在前面的AccountDao接口中，声明findAccountById()方法，通过id查询单个账户信息；声明findAllAccount()方法，用于查询所有账户信息。

```
// 通过id查询
public Account findAccountById(int id);
// 查询所有账户
public List<Account> findAllAccount();
```

4.2.3 query()方法

STEP 03

实现查询方法：在前面的的AccountDaoImpl类中，实现AccountDao接口中的findAccountById()方法和findAllAccount()方法，并调用query()方法分别进行查询。

```
// 这里只展示了其中一个方法，通过id查询单个账户信息
public Account findAccountById(int id) {
    //定义SQL语句
    String sql = "select * from account where id = ?";
    // 创建一个新的BeanPropertyRowMapper对象
    RowMapper<Account> rowMapper =
            new BeanPropertyRowMapper<Account>(Account.class);
    // 将id绑定到SQL语句中，通过RowMapper返回Object类型的单行记录
    return this.jdbcTemplate.queryForObject(sql, rowMapper, id);
}
```

4.2.3 query()方法

STEP 04

测试条件查询：创建测试类FindAccountByIdTest，用于测试条件查询。

```
public class FindAccountByIdTest {
    public static void main(String[] args) {
        // 加载配置文件
        ApplicationContext applicationContext =new
            ClassPathXmlApplicationContext("applicationContext.xml");
        // 获取AccountDao实例
        AccountDao accountDao =
                (AccountDao) applicationContext.getBean("accountDao");
        Account account = accountDao.findAccountById(1);
        System.out.println(account);       }
}
```

4.2.3 query()方法

STEP 05

测试查询所有用户信息：创建测试类FindAllAccountTest，用于查询所有用户账户信息。

```
public class FindAllAccountTest {
    public static void main(String[] args) {
        // 加载配置文件
        ApplicationContext applicationContext =new
            ClassPathXmlApplicationContext("applicationContext.xml");
        // 获取AccountDao实例
        AccountDao accountDao =
                (AccountDao) applicationContext.getBean("accountDao");
        List<Account> account = accountDao.findAllAccount(); // 执行方法
        for (Account act : account) {// 循环输出集合中的对象
            System.out.println(act); } }
}
```

4.2.3 query()方法

STEP 06

查看运行结果：在IDEA中启动FindAllAccountTest类，控制台会输出结果。

4.3 Spring事务管理概述

4.3.1 事务管理的核心接口

- 熟悉事务管理的核心接口，能够说出它的3个核心接口及内容

4.3.1 事务管理的核心接口

spring-tx-5.2.8.RELEAS依赖包的3个接口

PlatformTransactionManager接口：可以根据属性管理事务。

TransactionDefinition接口：用于定义事务的属性。

TransactionStatus接口：用于界定事务的状态 。

4.3.1 事务管理的核心接口

PlatformTransactionManager接口

PlatformTransactionManager 接口主要用于管理事务，该接口中提供了三个管理事物的方法。

方法	说明
TransactionStatus	用于获取事务状态信息
void commit(TransactionStatus status)	用于提交事务
void rollback(TransactionStatus status)	用于回滚事务

4.3.1 事务管理的核心接口

TransactionDefinition接口

TransactionDefinition接口中定义了事务描述相关的常量，其中包括了事务的隔离级别、事务的传播行为、事务的超时时间和是否为只读事务。

4.3.1 事务管理的核心接口

事务的隔离级别

隔离级别	说明
ISOLATION_DEFAULT	采用当前数据库默认的事务隔离级别。
ISOLATION_READ_UNCOMMITTED	读未提交。允许另外一个事务读取到当前未提交的数据，隔离级别最低，可能会导致脏读、幻读或不可重复读。
ISOLATION_READ_COMMITTED	读已提交。被一个事务修改的数据提交后才能被另一个事务读取，可以避免脏读，无法避免幻读，而且不可重复读。
ISOLATION_ REPEATABLE_READ	允许重复读，可以避免脏读，资源消耗上升。这是MySQL数据库的默认隔离级别。
REPEATABLE_SERIALIZABLE	事务串行执行，也就是按照时间顺序一一执行多个事务，不存在并发问题，最可靠，但性能与效率最低。

4.3.1 事务管理的核心接口

事务的传播行为

事务的传播行为是指处于不同事务中的方法在相互调用时，方法执行期间，事务的维护情况。例如，当一个事务的方法B调用另一个事务的方法A时，可以规定A方法继续在B方法所属的现有事务中运行，也可以规定A方法开启一个新事务，在新事务中运行，B方法所属的现有事务先挂起，等A方法的新事务执行完毕后再恢复。

4.3.1 事务管理的核心接口

TransactionDefinition接口中定义的7种事务传播行为

传播行为	说明
PROPAGATION_REQUIRED	默认的事务传播行为。如果当前存在一个事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
PROPAGATION_SUPPORTS	读未提交。允许如果当前存在一个事务，则加入该事务；如果当前没有事务，则以非事务方式执行。
PROPAGATION_MANDATORY	当前必须存在一个事务，如果没有，就抛出异常。
PROPAGATION_REQUIRES_NEW	创建一个新的事务，如果当前已存在一个事务，将已存在的事务挂起。
PROPAGATION_NOT_SUPPORTED	不支持事务，在没有事务的情况下执行，如果当前已存在一个事务，则将已存在的事务挂起。
ROPAGATION_NEVER	永远不支持当前事务，如果当前已存在一个事务，则抛出异常。
PROPAGATION_NESTED	如果当前存在事务，则在当前事务的一个子事务中执行。

4.3.1 事务管理的核心接口

事务的超时时间

事务的超时时间是指事务执行的时间界限，超过这个时间界限，事务将会回滚。

TransactionDefinition接口提供了TIMEOUT_DEFAULT常量定义事务的超时时间。

4.3.1 事务管理的核心接口

事务是否只读

当事务为只读时，该事务不修改任何数据，只读事务有助于提升性能，如果在只读事务中修改数据，会引发异常。

TransactionDefinition接口中除了提供事务的隔离级别、事务的传播行为、事务的超时时间和是否为只读事务的常量外，还提供了一系列方法来获取事务的属性。

4.3.1 事务管理的核心接口

TransactionDefinition接口常用方法

方法	说明
int getPropagationBehavior()	返回事务的传播行为
int getIsolationLevel()	返回事务的隔离层次
int getTimeout()	返回事务的超时属性
boolean isReadOnly()	判断事务是否为只读
String getName()	返回定义的事务名称

4.3.1 事务管理的核心接口

TransactionStatus接口

方法	说明
boolean isNewTransaction()	判断当前事务是否为新事务
boolean hasSavepoint()	判断当前事务是否创建了一个保存点
boolean isRollbackOnly()	判断当前事务是否被标记为rollback-only
void setRollbackOnly()	将当前事务标记为rollback-only
boolean isCompleted()	判断当前事务是否已经完成（提交或回滚）
void flush()	刷新底层的修改到数据库

4.3.2 事务管理的方式

- 熟悉事务管理的方式，能够说出Spring事务管理的两种方式分别是什么

4.3.2 事务管理的方式

Spring中的事务管理的两种方式

Spring中的事务管理分为两种方式，一种是传统的编程式事务管理，另一种是声明式事务管理。

编程式事务管理：通过编写代码实现的事务管理，包括定义事务的开始、正常执行后的事务提交和异常时的事务回滚。

声明式事务管理：通过AOP技术实现的事务管理，其主要思想是将事务管理作为一个“切面”代码单独编写，然后通过AOP技术将事务管理的“切面”代码植入到业务目标类中。

4.4 声明式事务管理

4.4.1 基于XML方式的声明式事务

- 掌握基于XML方式的声明式事务，能够在xml置文件中配置事务的相关声明

4.4.1 基于XML方式的声明式事务

如何实现XML方式的声明式事务

基于XML方式的声明式事务管理是通过在配置文件中配置事务规则的相关声明来实现的。在使用XML文件配置声明式事务管理时，首先要引入tx命名空间，在引入tx命名空间之后，可以使用<tx:advice>元素来配置事务管理的通知，进而通过Spring AOP实现事务管理。

配置<tx:advice>元素时，通常需要指定 id 和 transaction-manager 属性，其中，id 属性是配置文件中的唯一标识，transaction-manager 属性用于指定事务管理器。除此之外，<tx:advice>元素还包含子元素<tx: attributes>，<tx:attributes>元素可配置多个<tx:method>子元素，<tx:method>子元素主要用于配置事务的属性。

4.4.1 基于XML方式的声明式事务

<tx:method>元素的常用属性

属性	说明
name	用于指定方法名的匹配模式，该属性为必选属性，它指定了与事务属性相关的方法名。
propagation	用于指定事务的传播行为。
isolation	用于指定事务的隔离级别。
read-only	用于指定事务是否只读。
timeout	用于指定事务超时时间。
rollback-for	用于指定触发事务回滚的异常类。
no-rollback-for	用于指定不触发事务回滚的异常类。

4.4.1 基于XML方式的声明式事务

接下来通过一个案例演示如何通过XML方式实现Spring的声明式事务管理。本案例以9.2小节的项目代码和数据表为基础，编写一个模拟银行转账的程序，要求在转账时通过Spring对事务进行控制。案例具体实现步骤如下。

STEP 01

导入依赖：在chapter09项目的pom.xml文件中加入aspectjweaver依赖包和aopalliance依赖包作为实现切面所需的依赖包。

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.6</version>
    <scope>runtime</scope>          </dependency>
<dependency>
    <groupId>aopalliance</groupId>
    <artifactId>aopalliance</artifactId>
    <version>1.0</version>          </dependency>
```

4.4.1 基于XML方式的声明式事务

STEP 02

定义Dao层方法：AccountDao接口中声明转账方法transfer()。

```
// 转账
public void transfer(String outUser,
                     String inUser,
                     Double money);
```

4.4.1 基于XML方式的声明式事务

STEP 03

实现Dao层方法：AccountDaoImpl实现类中实现AccountDao接口中的transfer()方法。

```
// 转账 inUser：收款人； outUser：汇款人； money：收款金额
public void transfer(String outUser, String inUser, Double money) {
    // 收款时，收款用户的余额=现有余额+所汇金额
    this.jdbcTemplate.update("update account set balance = balance +? "
            + "where username = ?",money, inUser);
    // 模拟系统运行时的突发性问题
    int i = 1/0;
    // 汇款时，汇款用户的余额=现有余额-所汇金额
    this.jdbcTemplate.update("update account set balance = balance-? "
            + "where username = ?",money, outUser);
}
```

4.4.1 基于XML方式的声明式事务

STEP 04

修改配置文件：修改chapter09项目的配置文件applicationContext.xml，添加命名空间等相关配置代码。

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:tx="http://www.springframework.org/schema/tx"
 “><!-- 引入命名空间，这里只列举了两个-->
   <!-- 1.配置数据源；2.配置JDBC模板；3.定义id为accountDao的Bean；前3步省略--><!-- 4.事务管理器，依赖于数据源 -->
   <bean id="transactionManager" class=
"org.springframework.jdbc.datasource.DataSourceTransactionManager"
>
       <property name="dataSource" ref="dataSource" /></bean>
</beans>
```

4.4.1 基于XML方式的声明式事务

STEP 05

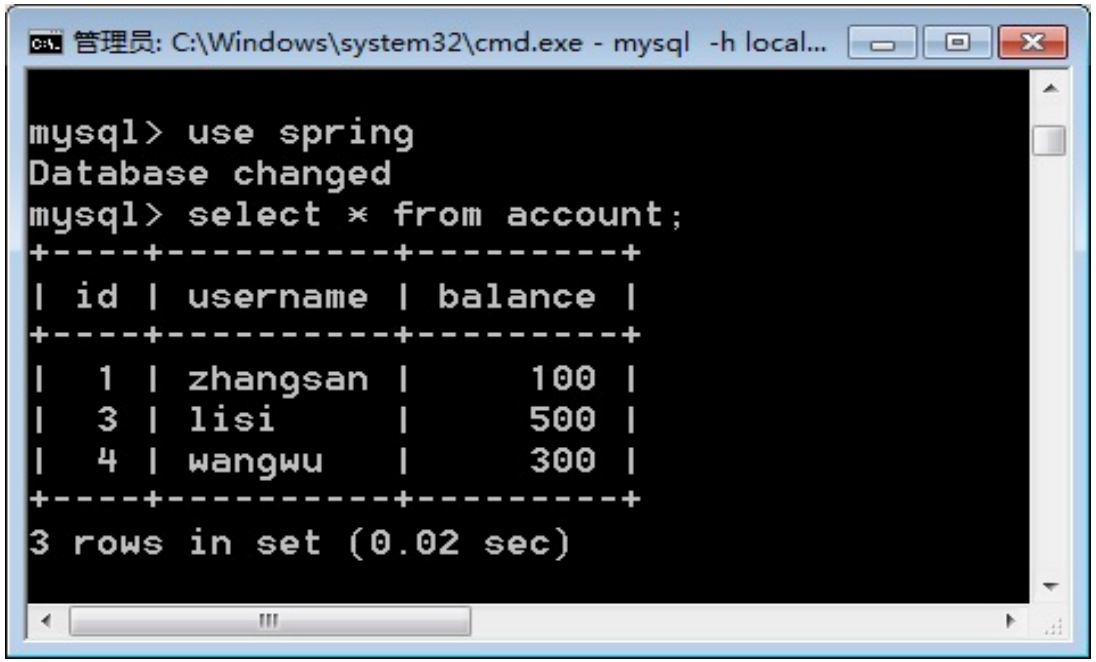
测试系统：创建测试类TransactionTest。

```
public class TransactionTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
        // 获取AccountDao实例
        AccountDao accountDao =
                (AccountDao)applicationContext.getBean("accountDao");
        // 调用实例中的转账方法
        accountDao.transfer("lisi", "zhangsan", 100.0);
        // 输出提示信息
        System.out.println("转账成功！");}}
```

4.4.1 基于XML方式的声明式事务

STEP 06

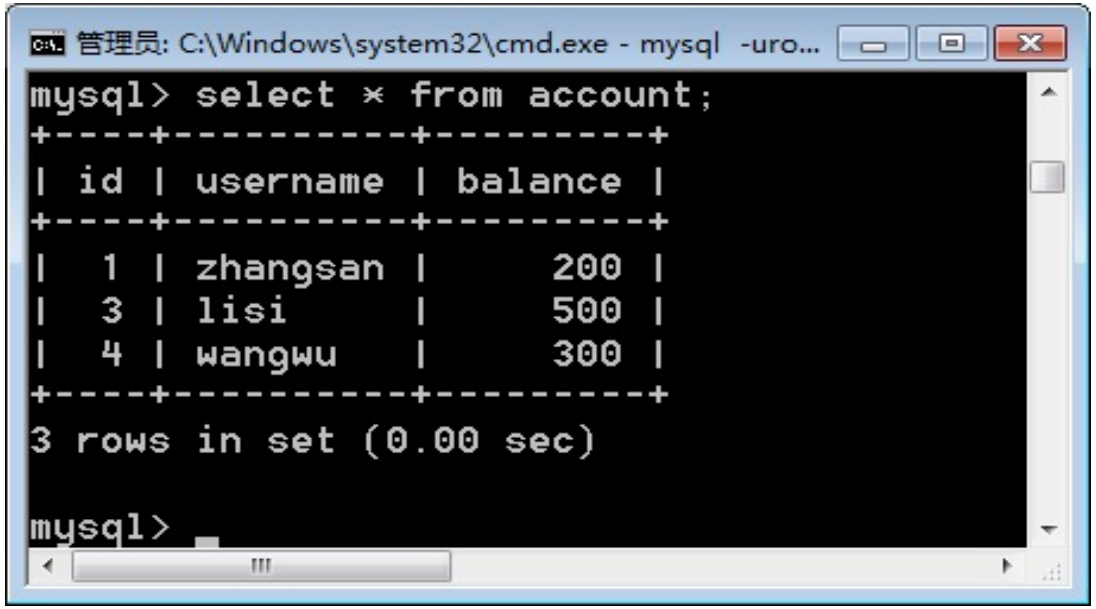
查看表数据：在执行转账操作前，也就是在执行第5不操作前，先查看account表中的数据。



4.4.1 基于XML方式的声明式事务

STEP 07

再查看表数据：在执行第5步后，控制台中报出了“/by zero”的算术异常信息。此时再次查询数据表account。



4.4.1 基于XML方式的声明式事务

未使用事物管理的缺陷

由XML方式实现声明式事务管理的案例可知，zhangsan的账户余额增加了100，而lisi的账户确没有任何变化，这样的情况显然是不合理的。这就是没有事务管理，系统无法保证数据的安全性与一致性，下面使用事务管理解决该问题。

4.4.1 基于XML方式的声明式事务

STEP 08

使用事务管理测试系统：在第4步的代码文件中添加事务管理的配置。

```
<!-- 5.编写通知，需要编写对切入点和具体执行事务细节-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"
                isolation="DEFAULT" read-only="false" /></tx:attributes>
</tx:advice>
<!-- 6.编写aop，使用AspectJ的表达式，让spring自动对目标生成代理-->
<aop:config>
    <aop:pointcut expression="execution(* com.itheima.*.*(..))"
            id="txPointCut" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"
</aop:config>
```

4.4.2 基于注解方式的声明式事务

- 熟悉基于注解方式的声明式事务，能够使用注解的方式配置事务的相关声明

4.4.2 基于注解方式的声明式事务

@Transactional的属性

属性	说明
value	用于指定使用的事务管理器
propagation	用于指定事务的传播行为
isolation	用于指定事务的隔离级别
timeout	用于指定事务的超时时间
readonly	用于指定事务是否为只读
rollbackFor	用于指定导致事务回滚的异常类数组
rollbackForClassName	用于指定导致事务回滚的异常类名称数组
noRollbackFor	用于指定不会导致事务回滚的异常类数组
noRollbackForClassName	用于指定不会导致事务回滚的异常类名称数组

4.4.2 基于注解方式的声明式事务

接下来对上一小节的案例进行修改，以注解方式来实现项目中的事务管理，具体实现步骤如下。

STEP 01

创建配置文件：创建配置文件applicationContext-annotation.xml，在该文件中声明事务管理器等配置信息。

```
<!– 前四步省略-->
<!-- 1.配置数据源：数据库驱动；连接数据库的url；连接数据库的用户名；连接数据库的密码 -->
<!-- 2.配置JDBC模板：默认必须使用数据源 -->
<!--3.定义id为accountDao的Bean：将jdbcTemplate注入到AccountDao实例中 -->
<!-- 4.事务管理器，依赖于数据源 -->
<!-- 5.注册事务管理器驱动 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

4.4.2 基于注解方式的声明式事务

STEP 02

修改Dao层实现类：在AccountDaoImpl类的transfer()方法上添加事务注解@Transactional。

```
@Transactional(propagation = Propagation.REQUIRED,
        isolation = Isolation.DEFAULT, readOnly = false)
public void transfer(String outUser, String inUser, Double money) {
    // 收款时，收款用户的余额=现有余额+所汇金额
    this.jdbcTemplate.update("update account set balance = balance +? "
        + "where username = ?",money, inUser);
    // 模拟系统运行时的突发性问题
    int i = 1/0;
    // 汇款时，汇款用户的余额=现有余额-所汇金额
    this.jdbcTemplate.update("update account set balance = balance-? "
        + "where username = ?",money, outUser);
}
```

4.4.2 基于注解方式的声明式事务

STEP 03

编写测试类：创建测试类AnnotationTest。

```
public class AnnotationTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext =new
   ClassPathXmlApplicationContext("applicationContext-
annotation.xml");
        // 获取AccountDao实例
        AccountDao accountDao =
                (AccountDao)applicationContext.getBean("accountDao");
        // 调用实例中的转账方法
        accountDao.transfer("lisi", "zhangsan", 100.0);
        // 输出提示信息
        System.out.println("转账成功！");}}
```

4.5 案例：实现用户登录

4.5 案例：实现用户登录

- 通过所学的Spring数据库编程知识，实现学生管理系统的用户登录功能。

4.5 案例：实现用户登录

案例要求

本案例要求学生在控制台输入用户名密码，如果用户账号密码正确则显示用户所属班级，如果登录失败则显示登录失败。实现用户登录项目运行成功后控制台效果如下所示。

4.5 案例：实现用户登录

思路分析

根据学生管理系统及其登录要求，可以分析案例的实现步骤如下。

（1）为了存储学生信息，需要创建一个数据库。

（2）为了程序连接数据库并完成对数据的增删改查操作，需要在XML配置文件中配置数据库连接和事务等信息。

（3）在Dao层实现查询用户信息的方法。

（4）在Controller层处理业务逻辑，如判断用户输入的用户名与密码是否正确 。

4.5 案例：实现用户登录

STEP 01

创建数据库：在MySQL中的spring数据库中创建一个名为student的表。

字段名	类型	长度	是否主键	说明
id	int	11	是	学生编号
username	varchar	255	否	学生姓名
password	varchar	255	否	学生密码
course	varchar	255	否	学生班级

4.5 案例：实现用户登录

STEP 02

编写实体类：创建Student类，在该类中定义id、username、password和course属性，以及属性对应的getter/setter方法。

```
public class Student {
    //学生ID
    private Integer id;
    //学生姓名
    private String username;
    //学生密码
    private String password;
    //学生班级
    private String course;
    // 省略getter/setter方法
}
```

4.5 案例：实现用户登录

STEP 03

编写配置文件：创建配置文件applicationContext-student.xml，在该文件中配置id为dataSource的数据源Bean和id为jdbcTemplate的JDBC模板Bean，并将数据源注入到JDBC模板中。

```
<!-- 1.配置数据源 -->
<!--数据库驱动 -->
<!--连接数据库的url -->
<!--连接数据库的用户名 -->
<!--连接数据库的密码 -->
<!-- 2.配置JDBC模板 -->
<!-- 默认必须使用数据源 -->
<!-- 3.定义id为accountDao的Bean -->
<!-- 将jdbcTemplate注入到AccountDao实例中 -->
<!-- 4.事务管理器，依赖于数据源 -->
<!-- 5.注册事务管理器驱动 -->
```

4.5 案例：实现用户登录

STEP 04

编写Dao层方法：创建StudentDao接口，在StudentDao接口中声明查询所有用户信息的方法。

```
package com.itheima.dao;
import com.itheima.entity.Student;
import java.util.List;
public interface StudentDao {
    //查询所有账户
    public List<Student> findAllStudent();
```

4.5 案例：实现用户登录

STEP 05

实现Dao层方法：创建StudentDaoImpl实现类，在StudentDaoImpl类中实现StudentDao接口中的findAllStudent()方法。

```
public class StudentDaoImpl implements StudentDao {
    // 声明JdbcTemplate属性，省略了setter方法
    private JdbcTemplate jdbcTemplate;
    public List<Student> findAllStudent() {
        String sql = "select * from student";
        RowMapper<Student> rowMapper =
                new BeanPropertyRowMapper<Student>(Student.class);
        // 执行静态的SQL查询，并通过RowMapper返回结果
        return this.jdbcTemplate.query(sql, rowMapper);
    }}
```

4.5 案例：实现用户登录

STEP 06

编写Controller层：创建StudentController类，用于实现用户登录操作。

```
public class StudentController {
    public static void main(String[] args) {
        Scanner sca=new Scanner(System.in);
        String name=sca.nextLine();
        ApplicationContext applicationContext =new
        ClassPathXmlApplicationContext("applicationContext-student.xml");
        // 获取AccountDao实例
        StudentDao studentDao =
                (StudentDao) applicationContext.getBean("studentDao");
        List<Student> student = studentDao.findAllStudent();
        // for循环输出集合中的对象，此处省略        }}
```

4.5 案例：实现用户登录

STEP 07

查看运行结果：在IDEA中启动StudentController类，在控制台按照提示输入账号密码进行登录。

本章小结

本章主要讲解了Spring的数据库编程。首先介绍了Spring JDBC，包括JdbcTemplate概述和Spring JDBC的配置；然后讲解了JdbcTemplate的增删改查操作，包括excute()方法、update()方法和query()方法；接着讲解了Spring事务管理概述，包括事务管理的核心接口和事务管理的方式；最后讲解了两种实现声明式事务管理的方式，基于XML方式的声明式事务和基于注解方式的声明式事务。通过本章的学习，读者可以对Spring的数据库编程有一定的了解，为框架中数据库开发奠定了基础。