# 1. Spring对JDBC的支持

Spring对JDBC提供了很好的支持，主要体现在：

（1）很好的支持C3P0数据库连接池；

（2）提供了JDBC模板，即JdbcTemplate，简化了JDBC的操作（类似于DBUtils）。

接下来就使用Spring管理数据库连接池和使用JdbcTemplate。步骤如下：

（1）首先要在项目中再引入Spring对数据库支持相关的包：

|  |
| --- |
| spring-jdbc-4.3.7.RELEASE.jar （JDBC支持包）  spring-tx-4.3.7.RELEASE.jar （数据库事物支持包） |

初次之外，还要导入JDBC驱动包，这是每个操作数据库的Java程序必须的。并且项目中整合了C3P0连接池，所以把C3P0包夜加入到项目中。

（2）分析。

由于使用Spring进行管理，我们只要能在程序中拿到JdbcTemplate对象，通过该对象进行数据库操作即可。而JdbcTemplate的创建依赖于数据源dataSource，而dataSource是从连接池中得到的。这样分析下来，我们主要配置的就是JdbcTemplate -> dataSource -> 数据库连接池配置。

案例是这样一个简单的案例：UserDao可拿到JDBCTemplate对象，即依赖JdbcTemplate，因此也要配置UserDao的依赖关系，我们直接在主类中得到UserDao对象操作即可。

具体配置如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 配置数据源对象。使用c3p0连接池 -->* <**bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  *<!-- 配置连接池属性 -->  <!-- 之所以能够配置这些属性，是因为ComboPooledDataSource类中原本就有这些属性 -->* <**property name="driverClass" value="com.mysql.jdbc.Driver"** />  <**property name="jdbcUrl" value="jdbc:mysql://localhost:3306/dbtest?useSSL=false&amp;useUnicode=true&amp;characterEncoding=utf8"** />  <**property name="user" value="root"** />  <**property name="password" value="123456"** />  <**property name="initialPoolSize" value="3"** />  <**property name="maxPoolSize" value="10"** />  <**property name="maxStatements" value="100"** />  <**property name="acquireIncrement" value="2"** />  </**bean**>  *<!-- 配置JdbcTemplate -->* <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  *<!-- 其中的dataSource属性就依赖上面的dataSource -->* <**property name="dataSource" ref="dataSource"** />  </**bean**>  *<!-- 配置UserDao -->* <**bean id="userDao" class="com.zhang.test.UserDao"**>  *<!-- 其中的属性又依赖上述的jdbcTemplate -->* <**property name="jdbcTemplate" ref="jdbcTemplate"** />  </**bean**> </**beans**> |

（3）UserDao等代码中就会很简洁。

UserDao代码例子（Student实体类自行编写）：

操作的student数据表，其中字段为：id（int型）、name（varchar型）和entranceTime（入学时间，datetime型）。

|  |
| --- |
| **package** com.zhang.test;  **import** org.springframework.jdbc.core.JdbcTemplate; **import** org.springframework.jdbc.core.RowMapper;  **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.util.Date; **import** java.util.List;  **public class** UserDao {  *// 依赖JdbcTemplate* **private** JdbcTemplate **jdbcTemplate**;  *// 需要提供setter方法* **public void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {  **this**.**jdbcTemplate** = jdbcTemplate;  }   *// 增加操作* **public void** add() {  **jdbcTemplate**.update(**"insert into student values(?, ?, ?)"**, 10, **"张三"**, **new** Date());  }   *// 查询全部学生* **public** List<Student> queryAll() **throws** SQLException {  *// // 因为此query()方法返回的直接是List集合，所以可直接返回* **return jdbcTemplate**.query(**"select** *\** **from student"**, **new** RowMapper<Student>() {  @Override  **public** Student mapRow(ResultSet rs, **int** i) **throws** SQLException {  *//这里的rs是当前行的结果集。index就是当前行  // 因此处理时无需调用rs.next()。直接处理好一行数据即可* Student student = **new** Student();  student.setId(rs.getInt(**"id"**));  student.setName(rs.getString(**"name"**));  // 要想获得数据库的datetime类型完整数据，要使用getTimestamp()获得时间戳Timestamp对象，该对象是Date的子类，因此这里可直接赋值  student.setEntranceTime(rs.getTimestamp(**"entranceTime"**));  **return** student;  }  });  }   *// 查询指定学生* **public** Student find(**int** id) {  List<Student> list = **jdbcTemplate**.query(**"select** *\** **from student where id = ?"**, **new** RowMapper<Student>() {  @Override  **public** Student mapRow(ResultSet rs, **int** i) **throws** SQLException {  Student student = **new** Student();  student.setId(rs.getInt(**"id"**));  student.setName(rs.getString(**"name"**));  student.setEntranceTime(rs.getDate(**"entranceTime"**));  **return** student;  }  }, id);  *// 返回时就要对list进行处理* **return** list.size() == 0 ? **null** : list.get(0);  } } |

可以看出，JDBCTemplate的用法和DBUtils用法很相似，增删改都是用upate()方法，查询用query()方法。只是JdbcTemplate中，处理查询的数据时使用的是RowMapper接口，从mapRow()方法中得到的是一行的数据，只需要处理一行数据，返回值是List集合。因此当返回单个对象时，就需要判断下size()是否为0。此外，由于这样的设计，mapRow()的针对同一个对象的处理代码基本是一致的（如上面的find和query方法），也可以直接把重复代码写在内部类中重复利用。

（4）主类测试：

|  |
| --- |
| **package** com.zhang.test;  **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **import** java.sql.SQLException; **import** java.util.List;  **public class** Demo {  **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *// 拿到UserDao对象* UserDao userDao = (UserDao)ac.getBean(**"userDao"**);  **try** {  List<Student> list = userDao.queryAll();  System.***out***.println(list);  Student student = userDao.find(123);  System.***out***.println(student); *// 无记录，显示null* } **catch** (SQLException ex) {  ex.printStackTrace();  }  } } |

# 2. 事物管理

## 2.1 什么是事物管理

从开发者角度来说，事物管理分为“编程式事物管理”和“声明式事物管理”。解释如下：

（1）编程式事物管理：

之前我们使用的都是编程式事物管理。编程式事物管理就是在代码中手动处理事物。比如在JDBC中可通过设置connection.setAutoCommit(false)来手动控制事物。在Hibernate中，我们调用的beginTransaction()和commit()方法就是手动处理事物。

这种事物可以对方法中某几行代码进行详细的事物控制，是细粒度的事物控制，比较灵活。缺点是开发过程中比较繁琐。

（2）声明式事物管理：

声明式事物管理就是对要控制的事物进行“声明”，比如可将事物控制配置在文件中（或者通过注解的方式配置），可按需更改或者移除。这种事物是粗粒度的事物控制，只能在方法上应用事物控制。

Spring中就提供了声明式事物管理，其核心是基于AOP给方法添加事物控制。Spring中用声明式事物管理器类具体的实现此功能。比如JDBC使用DataSourceTransactionManager事物管理器；Hibernate使用HibernateTransactionManager事物管理器。

## 2.2 三层结构中的事物管理

三层结构中，访问的数据的层次为：表示层 -> 业务逻辑层（Service） -> 数据访问层 -> 数据库。若采用声明式事物管理，那么必定要将事物声明配置在Service层中。这是因为一项业务的操作可能涉及多个DAO对象，若此项业务成功执行，则要求其中的DAO操作全部执行成功，否则其中的DAO操作都将操作失败。所以需要将生命式事物管理配置在Service层中。

## 2.3 声明式事物管理案例

### 2.3.1 XML实现声明式事物管理

基于第一章的例子，添加Service层，主要在此层实现声明式事物管理。

在Service层中，save2()方法需要一次保存两个学生信息，这是一个事物，我们需要在Service层配置事物，使得两个学生要么全部插入成功，要么全部失败。

先修改之前UserDao的add()方法，即接收一个学生参数：

|  |
| --- |
| *// 增加操作* **public void** add(Student student) {  **jdbcTemplate**.update(**"insert into student values(?, ?, ?)"**, student.getId(), student.getName(), student.getEntranceTime()); } |

Service层代码如下：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.Date;  **public class** UserService {  *// 依赖UserDao* **private** UserDao **userDao**;  *// setter方法* **public void** setUserDao(UserDao userDao) {  **this**.**userDao** = userDao;  }   *// save2()方法* **public void** save2() {  *// 保存第一个学生* **userDao**.add(**new** Student(12, **"张三"**, **new** Date()));  **int** a = 9 / 0; *// 故意让程序出错* **userDao**.add(**new** Student(13, **"李四"**, **new** Date()));  } } |

之后主类测试就应使用UserService：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *// 拿到UserDao对象* UserService userService = (UserService)ac.getBean(**"userService"**);  userService.save2(); } |

bean.xml需要添加UserService配置以及事物配置（其余配置无需变动）：

|  |
| --- |
| *<!-- service配置 -->* <**bean id="userService" class="com.zhang.test.UserService"**>  <**property name="userDao" ref="userDao"** /> </**bean**> *<!-- 事物配置。要使用tx名称空间 --> <!-- 1. 配置事务管理器类，这里配置JDBC的事务管理器 -->* <**bean id="txJDBC" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!-- 其中引用下数据源 -->* <**property name="dataSource" ref="dataSource"** /> </**bean**> *<!-- 2. 配置事物管理器，就是如何管理事物 --> <!-- transaction-manager就是上面的txJDBC -->* <**tx:advice id="txAdvice" transaction-manager="txJDBC"**>  <**tx:attributes**>  *<!-- attributes节点中配置详细。name指定对哪些方法应用事物，这里用了通配符 -->  <!-- read-only表示此事物是否为只读。由于save2()这个方法是添加数据，因此不能设置为true。对于查询操作，可设置为true -->* <**tx:method name="\*" read-only="false"** />  </**tx:attributes**> </**tx:advice**> *<!-- 3.AOP配置。指定切入点，这样Spring才能为下面的方法植入上述的事物增强配置的相关代码（相关代码由Spring提供）。 -->* <**aop:config**>  *<!-- 切入点表达式 -->* <**aop:pointcut id="pt" expression="execution(\* com.zhang.test.UserService.save2(..))"** />  *<!-- 增强事物和切入点的关联配置 -->* <**aop:advisor advice-ref="txAdvice" pointcut-ref="pt"**/> </**aop:config**> |

主要注意的是，tx名称空间引用的是“xmlns:tx="http://www.springframework.org/schema/tx"”，因为还有一个名称空间也叫tx，但是不是用来配置事物的，用错了就会出错。

这样我们运行程序，Service中执行了第一个add()方法后出现异常，但由于我们配置了事物，因此数据库中并不会增加数据。若我们把int a = 9 / 0; 这行代码去掉，则程序会成功添加两个学生数据。

### 2.3.2 注解实现声明式事物管理

使用注解可用于简化配置。这里只需要在以前的基础上添加注解即可。

使用注解不意味着完全抛弃XML，首先在DAO和Service中，要使用@Controller和@Resource注解；使用@Transactional注解配置声明式事物；但是对于dataSource、jdbcTemplate等还是在xml中配置。例如：

UserDao：

|  |
| --- |
| **package** com.zhang.test;  @Repository *// Dao层的“@Component”* **public class** UserDao {  @Autowired  **private** JdbcTemplate jdbcTemplate;   *// 增加操作* **public void** add(Student student) {  jdbcTemplate.update(**"insert into student values(?, ?, ?)"**, student.getId(), student.getName(), student.getEntranceTime());  }   *// 查询全部学生* **public** List<Student> queryAll() **throws** SQLException {  **return** jdbcTemplate.query(**"select** *\** **from student"**, **new** RowMapper<Student>() {  @Override  **public** Student mapRow(ResultSet rs, **int** i) **throws** SQLException {  Student student = **new** Student();  student.setId(rs.getInt(**"id"**));  student.setName(rs.getString(**"name"**));  student.setEntranceTime(rs.getTimestamp(**"entranceTime"**));  **return** student;  }  });  } } |

UserService中，注意使用的@Transactional注解：

|  |
| --- |
| **package** com.zhang.test;  **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Service; **import** org.springframework.transaction.annotation.Isolation; **import** org.springframework.transaction.annotation.Transactional;  **import** java.sql.SQLException; **import** java.util.Date; **import** java.util.List;  @Service **public class** UserService {  @Autowired  **private** UserDao **userDao**;   @Transactional(  readOnly = **false**, *// 只读为false，因为这是添加* timeout = -1, *// 设置事物的超时时间。设置为-1，就是不限制超时时间* isolation = Isolation.***DEFAULT***, *// 事物的隔离级别，这是默认的* noRollbackFor = ArithmeticException.**class** *// 设置遇到数学异常不回滚* )  **public void** save2() {  **userDao**.add(**new** Student(14, **"张三"**, **new** Date()));  **int** a = 9 / 0;  **userDao**.add(**new** Student(13, **"李四"**, **new** Date()));  }   @Transactional(readOnly = **true**) *// 这个可设置为只读* **public** List<Student> getStudents() {  List<Student> list = **null**;  **try** {  list = **userDao**.queryAll();  *// 因为配置了此事物为只读。因此下面的方法并不能真正操作数据库，数据库不会有数据添加，虽然程序不会出错。* **userDao**.add(**new** Student(100, **"测试"**, **new** Date())); *// 不会添加数据* } **catch** (SQLException ex) {  ex.printStackTrace();  } **finally** {  **return** list;  }  } } |

主类测试：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *// 拿到UserDao对象* UserService userService = (UserService)ac.getBean(**"userService"**);  *// 读取数据* List<Student> studentList = userService.getStudents();  System.***out***.println(studentList);  userService.save2();  } } |

程序执行save2方法时，会只添加“14 张三”这个记录，这并不是因为我们配置的事物不对，而是因为我们配置了“产生数学异常不进行回滚”，所以说异常之前的数据能够插入成功。

说明：@Transactional注解定义在方法上时，是对当前方法应用事物，而定义到类上时，是对类中所有的方法都应用此事物。此外，@Transactional还有一个常用的属性：propagation，可以指定事物的传播行为。此属性的默认值是“ Propagation.REQUIRED”，意思是如果存在一个事务，则支持当前事务。如果没有事务则开启。怎么理解呢，可以和“Propagation.REQUIRES\_NEW”这个属性值作比较。Propagation.REQUIRES\_NEW表示总是开启一个新的事务。如果一个事务存在，则将这个存在的事务挂起。

意思就是REQUIRES\_NEW时，如果当前运行的方法中，会调用其他方法，而被调用的方法也有事务，那么当前事务会挂起，被调用方法会重新开启一个新的事务，该事务执行完后，挂起的事务才继续执行。也就是事物之间互不干扰。而默认的REQUIRED则会将所有的执行看作是一个整体。

有关具体内容还可网络搜索学习。