作者博客：http://jinnianshilongnian.iteye.com/ 欢迎访问

第八章　对 ORM 的支持

8.1　概述

8.1.1　ORM 框架

ORM 全称对象关系映射（Object/Relation Mapping），指将 Java 对象状态自动映射到关系数据库中的数据上，从而提供透明化的持久化支持，即把一种形式转化为另一种形式。

对象与关系数据库之间是不匹配，我们把这种不匹配称为阻抗失配，主要表现在：

- 关系数据库首先不支持面向对象技术如继承、多态，如何使关系数据库支持它们；
- 关系数据库是由表来存放数据，而面向对象使用对象来存放状态；其中表的列称为属性，而对象的属性就是属性，因此需要通过解决这种不匹配；
- 如何将对象透明的持久化到关系数据库表中；
- 如果一个对象存在横跨多个表的数据，应该如何为对象建模和映射。

其中这些阻抗失配只是其中的一小部分，比如还有如何将 SQL 集合函数结果集映射到对象，如何在对象中处理主键等。

ORM 框架就是用来解决这种阻抗失配，提供关系数据库的对象化支持。

ORM 框架不是万能的，同样符合 **80/20 法则**，应解决的最核心问题是如何在关系数据库表中的行和对象进行映射，并自动持久化对象到关系数据库。

ORM 解决方案适用于解决透明持久化、小结果集查询等；对于复杂查询，大结果集数据处理还是没有任何帮助的。

目前已经有许多 ORM 框架产生，如 Hibernate、JDO、JPA、iBATIS 等等，这些 ORM 框架各有特色，Spring 对这些 ORM 框架提供了很好的支持，接下来首先让我们看一下 Spring 如何支持这些 ORM 框架。

8.1.2　Spring 对 ORM 的支持

Spring 对 ORM 的支持主要表现在以下方面：

- 一致的异常体系结构，对第三方 ORM 框架抛出的专有异常进行包装，从而在使我们在 Spring 中只看到 DataAccessException 异常体系；
- 一致的 DAO 抽象支持：提供类似与 JdbcSupport 的 DAO 支持类

HibernateDaoSupport，使用 HibernateTemplate 模板类来简化常用操作，HibernateTemplate 提供回调接口来支持复杂操作；

- Spring 事务管理：Spring 对所有数据访问提供一致的事务管理，通过配置方式，简化事务管理。

Spring 还在测试、数据源管理方面提供支持，从而允许方便测试，简化数据源使用。

接下来让我们学习一下 Spring 如何集成 ORM 框架—Hibernate。

8.2 集成 Hibernate3

Hibernate 是全自动的 ORM 框架，能自动为对象生成相应 SQL 并透明的持久化对象到数据库。

Spring2.5+版本支持 Hibernate 3.1+版本，不支持低版本，Spring3.0.5 版本提供对 Hibernate 3.6.0 Final 版本支持。

8.2.1 如何集成

Spring 通过使用如下 Bean 进行集成 Hibernate：

- LocalSessionFactoryBean ：用于支持 XML 映射定义读取：
  - configLocation 和 configLocations：用于定义 Hibernate 配置文件位置，一般使用如 classpath:hibernate.cfg.xml 形式指定；
  - mappingLocations ：用于指定 Hibenate 映射文件位置，如 chapter8/hbm/user.hbm.xml；
  - hibernateProperties：用于定义 Hibernate 属性，即 Hibernate 配置文件中的属性；
  - dataSource：定义数据源；

  hibernateProperties、dataSource 用于消除 Hibernate 配置文件，因此如果使用 configLocations 指定配置文件，就不要设置这两个属性了，否则会产生重复配置。推荐使用 dataSource 来指定数据源，而使用 hibernateProperties 指定 Hibernate 属性。
- AnnotationSessionFactoryBean：用于支持注解风格映射定义读取，该类继承 LocalSessionFactoryBean 并额外提供自动查找注解风格配置模型的能力：
  - annotatedClasses：设置注解了模型类，通过注解指定映射元数据。
  - packagesToScan：通过扫描指定的包获取注解模型类，而不是手工指定，如“cn.javass.**.model”将扫描 cn.javass 包及子包下的 model 包下的所有注解模型类。

接下来学习一下 Spring 如何集成 Hibernate 吧：

1、准备 jar 包：

首先准备 Spring 对 ORM 框架支持的 jar 包：

```
org.springframework.orm-3.0.5.RELEASE.jar   //提供对 ORM 框架集成
```

下载 hibernate-distribution-3.6.0.Final 包，获取如下 Hibernate 需要的 jar 包：

```
hibernate3.jar                                  //核心包
lib\required\antlr-2.7.6.jar                    //HQL 解析时使用的包
lib\required\javassist-3.9.0.GA.jar             //字节码类库，类似于 cglib
lib\required\commons-collections-3.1.jar        //对集合类型支持包，前边测试时已
                                                  经提供过了，无需再拷贝该包了
lib\required\dom4j-1.6.1.jar                    //xml 解析包，用于解析配置使用
lib\required\jta-1.1.jar                        //JTA 事务支持包
lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar //用于支持 JPA
```

下载 slf4j-1.6.1.zip（http://www.slf4j.org/download.html），slf4j 是日志系统门面（Simple Logging Facade for Java），用于对各种日志框架提供给一致的日志访问接口，从而能随时替换日志框架（如 log4j、java.util.logging）：

```
slf4j-api-1.6.1.jar                 //核心 API
slf4j-log4j12-1.6.1.jar             //log4j 实现
```

将这些 jar 包添加到类路径中。

2、 对象模型定义，此处使用第七章中的 UserModel：

```
package cn.javass.spring.chapter7;
public class UserModel {
    private int id;
    private String myName;
    //省略getter和setter
}
```

3、 Hibernate 映射定义（chapter8/hbm/user.hbm.xml），定义对象和数据库之间的映射：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="cn.javass.spring.chapter7.UserModel" table="test">
        <id name="id" column="id"><generator class="native"/></id>
        <property name="myName" column="name"/>
    </class>
</hibernate-mapping>
```

4、 数据源定义，此处使用第 7 章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

5、 SessionFactory 配置定义（chapter8/applicationContext-hibernate.xml）：

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
      <property name="dataSource" ref="dataSource"/> <!-- 指定数据源 -->
        <property name="mappingResources">        <!-- 指定映射定义 -->
          <list>
            <value>chapter8/hbm/user.hbm.xml</value>
          </list>
        </property>
        <property name="hibernateProperties">     <!--指定Hibernate属性 -->
          <props>
            <prop key="hibernate.dialect">
              org.hibernate.dialect.HSQLDialect
            </prop>
          </props>
        </property>
</bean>
```

6、 获取 SessionFactory：

```
package cn.javass.spring.chapter8;
//省略import
public class HibernateTest {
    private static SessionFactory sessionFactory;
    @BeforeClass
    public static void beforeClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-hibernate.xml"};
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(configLocations);
        sessionFactory = ctx.getBean("sessionFactory", SessionFactory.class);
    }
}
```

此处我们使用了 chapter7/applicationContext-resources.xml 定义的“dataSource”数

据源，通过 ctx.getBean("sessionFactory", SessionFactory.class)获取 SessionFactory。

7、 通过 SessionFactory 获取 Session 对象进行创建和删除表：

```
@Before
public void setUp() {
   //id自增主键从0开始
   final String createTableSql = "create memory table test" +
      "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
      "name varchar(100))";
      sessionFactory.openSession().
            createSQLQuery(createTableSql).executeUpdate();
}
@After
public void tearDown() {
      final String dropTableSql = "drop table test";
      sessionFactory.openSession().
            createSQLQuery(dropTableSql).executeUpdate();
}
```

使用 SessionFactory 创建 Session，然后通过 Session 对象的 createSQLQuery 创建本地 SQL 执行创建和删除表。

8、 使用 SessionFactory 获取 Session 对象进行持久化数据：

```
@Test
public void testFirst() {
      Session session = sessionFactory.openSession();
      Transaction transaction = null;
      try {
            transaction = beginTransaction(session);
            UserModel model = new UserModel();
            model.setMyName("myName");
            session.save(model);
      } catch (RuntimeException e) {
            rollbackTransaction(transaction);
            throw e;
      } finally {
            commitTransaction(session);
      }
}
```

```
private Transaction beginTransaction(Session session) {
        Transaction transaction = session.beginTransaction();
        transaction.begin();
        return transaction;
}
private void rollbackTransaction(Transaction transaction) {
        if(transaction != null) {
                transaction.rollback();
        }
}
private void commitTransaction(Session session) {
        session.close();
}
```

使用 SessionFactory 获取 Session 进行操作，必须自己控制事务，而且还要保证各个步骤不会出错，有没有更好的解决方案把我们从编程事务中解脱出来？Spring 提供了 HibernateTemplate 模板类用来简化事务处理和常见操作。

8.2.2 使用 HibernateTemplate

HibernateTimplate 模板类用于简化事务管理及常见操作，类似于 JdbcTemplate 模板类，对于复杂操作通过提供 HibernateCallback 回调接口来允许更复杂的操作。

接下来示例一下 HibernateTemplate 的使用：

```
@Test
public void testHibernateTemplate() {
        HibernateTemplate hibernateTemplate =
                new HibernateTemplate(sessionFactory);
        final UserModel model = new UserModel();
        model.setMyName("myName");
        hibernateTemplate.save(model);
        //通过回调允许更复杂操作
        hibernateTemplate.execute(new HibernateCallback<Void>() {
                @Override
                public Void doInHibernate(Session session)
                        throws HibernateException, SQLException {
                        session.save(model);
                        return null;
                }});
}
```

通过 new HibernateTemplate(sessionFactory) 创建 HibernateTemplate 模板类对象，通过调用模板类的 save 方法持久化对象，并且自动享受到 Spring 管理事务的好处。

而且 HibernateTemplate 提供使用 HibernateCallback 回调接口的方法 execute 用来支持复杂操作，当然也自动享受到 Spring 管理事务的好处。

8.2.3 集成 Hibernate 及最佳实践

类似于 JdbcDaoSupport 类，Spring 对 Hibernate 也提供了 HibernateDaoSupport 类来支持一致的数据库访问。HibernateDaoSupport 也是 DaoSupport 实现：

接下来示例一下 Spring 集成 Hibernate 的最佳实践：

1、 定义 Dao 接口，此处使用 cn.javass.spring.chapter7.dao. IUserDao：

2、 定义 Dao 接口实现，此处是 Hibernate 实现：

```
package cn.javass.spring.chapter8.dao.hibernate;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import cn.javass.spring.chapter7.UserModel;
import cn.javass.spring.chapter7.dao.IUserDao;
public class UserHibernateDaoImpl extends HibernateDaoSupport
        implements IUserDao {
        private static final String COUNT_ALL_HQL =
                "select count(*) from UserModel";
        @Override
        public void save(UserModel model) {
                getHibernateTemplate().save(model);
        }
        @Override
        public int countAll() {
                Number count =
                   (Number) getHibernateTemplate().find(COUNT_ALL_HQL).get(0);
                return count.intValue();
        }
}
```

此处注意首先 Hibernate 实现放在 dao.hibernate 包里，其次实现类命名如 UserHibernateDaoImpl，即×××HibernateDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

3、进行资源配置，使用 resources/chapter7/applicationContext-resources.xml：

4、dao 定义配置，在 chapter8/applicationContext-hibernate.xml 中添加如下配置：

```
<bean id="abstractDao" abstract="true">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="userDao"
    class="cn.javass.spring.chapter8.dao.hibernate.UserHibernateDaoImpl"
    parent="abstractDao"/>
```

首先定义抽象的 abstractDao，其有一个 sessionFactory 属性，从而可以让继承的子类自动继承 sessionFactory 属性注入；然后定义 userDao，且继承 abstractDao，从而继承 sessionFactory 注入；我们在此给配置文件命名为 applicationContext-hibernate.xml 表示 Hibernate 实现。

5、最后测试一下吧（cn.javass.spring.chapter8. HibernateTest）：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-hibernate.xml"};
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和 Spring JDBC 框架的最佳实践完全一样，除了使用 applicationContext-hibernate.xml 代替了 applicationContext-jdbc.xml，其他完全一样。也就是说，DAO 层的实现替换可以透明化。

8.2.4 Spring+Hibernate 的 CRUD

Spring+Hibernate CRUD（增删改查）我们使用注解类来示例，让我们看具体示例吧：

1、首先定义带注解的模型对象 UserModel2：

- 使用 JPA 注解@Table 指定表名映射；
- 使用注解@Id 指定主键映射；
- 使用注解@ Column 指定数据库列映射；

```
package cn.javass.spring.chapter8;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "test")
public class UserModel2 {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(name = "name")
    private String myName;
    //省略 getter 和 setter
}
```

2、 定义配置文件（chapter8/applicationContext-hibernate2.xml）：

2.1、 定义 SessionFactory：

此处使用 AnnotationSessionFactoryBean 通过 annotatedClasses 属性指定注解模型来定义映射元数据；

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.AnnotationSession
      FactoryBean">
  <property name="dataSource" ref="dataSource"/>   <!-- 1、指定数据源  -->
  <property name="annotatedClasses">                <!-- 2、指定注解类  -->
      <list><value>cn.javass.spring.chapter8.UserModel2</value></list>
  </property>
  <property name="hibernateProperties"><!-- 3、指定Hibernate属性  -->
    <props>
      <prop key="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
      </prop>
    </props>
  </property>
</bean>
```

2.2、 定义 HibernateTemplate ：

```
<bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate3.HibernateTemplate">
   <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

3、最后进行 CURD 测试吧：

```
@Test
public void testCURD() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-hibernate2.xml"};
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext(configLocations);
    HibernateTemplate hibernateTemplate =
        ctx.getBean(HibernateTemplate.class);
    UserModel2 model = new UserModel2();
    model.setMyName("test");
    insert(hibernateTemplate, model);
    select(hibernateTemplate, model);
    update(hibernateTemplate, model);
    delete(hibernateTemplate, model);
}
```

```
private void insert(HibernateTemplate hibernateTemplate, UserModel2 model) {
    hibernateTemplate.save(model);
}
private void select(HibernateTemplate hibernateTemplate, UserModel2 model) {
    UserModel2 model2 = hibernateTemplate.get(UserModel2.class, 0);
    Assert.assertEquals(model2.getMyName(), model.getMyName());
    List<UserModel2> list = hibernateTemplate.find("from UserModel2");
    Assert.assertEquals(list.get(0).getMyName(), model.getMyName());
}
private void update(HibernateTemplate hibernateTemplate, UserModel2 model) {
    model.setMyName("test2");
    hibernateTemplate.update(model);
}
private void delete(HibernateTemplate hibernateTemplate, UserModel2 model) {
    hibernateTemplate.delete(model);
}
```

Spring 集成 Hibernate 进行增删改查是不是比 Spring JDBC 方式简单许多，而且支持注解方式配置映射元数据，从而减少映射定义配置文件数量。

8.3 集成 iBATIS

iBATIS 是一个半自动化的 ORM 框架，需要通过配置方式指定映射 SQL 语句，而不是由框架本身生成（如 Hibernate 自动生成对应 SQL 来持久化对象），即 Hibernate 属于全自动 ORM 框架。

Spring 提供对 iBATIS 2.X 的集成，提供一致的异常体系、一致的 DAO 访问支持、Spring 管理事务支持。

Spring 2.5.5+版本支持 iBATIS 2.3+版本，不支持低版本。

8.3.1 如何集成

Spring 通过使用如下 Bean 进行集成 iBATIS：

- SqlMapClientFactoryBean：用于集成 iBATIS。
  - configLocation 和 configLocations：用于指定 SQL Map XML 配置文件，用于指定如数据源等配置信息；
  - mappingLocations：用于指定 SQL Map 映射文件，即半自动概念中的 SQL 语句定义；
  - sqlMapClientProperties：定义 iBATIS 配置文件配置信息；
  - dataSource：定义数据源。

  如果在 Spring 配置文件中指定了 DataSource，就不要在 iBATIS 配置文件指定了，否则 Spring 配置文件指定的 DataSource 将覆盖 iBATIS 配置文件中定义的 DataSource。

接下来示例一下如何集成 iBATIS：

1、准备需要的 jar 包，从 spring-framework-3.0.5.RELEASE-dependencies.zip 中拷贝如下 jar 包：

com.springsource.com.ibatis-2.3.4.726.jar

2、 对象模型定义，此处使用第七章中的 UserModel；

3、 iBATIS **映射定义（chapter8/sqlmaps/UserSQL.xml）：**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
      "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="UserSQL">
   <statement id="createTable">
      <!--id自增主键从0开始  -->
      <![CDATA[
            create memory table test(
               id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
               name varchar(100))
      ]]>
   </statement>
   <statement id="dropTable">
      <![CDATA[    drop table test    ]]>
   </statement>
   <insert id="insert" parameterClass="cn.javass.spring.chapter7.UserModel">
      <![CDATA[
         insert into test(name) values (#myName#)
      ]]>
      <selectKey resultClass="int" keyProperty="id" type="post">
         <!--  获取hsqldb插入的主键  -->
         call identity();
         <!-- mysql使用select last_insert_id();获取插入的主键  -->
      </selectKey>
   </insert>
</sqlMap>
```

4、 iBATIS 配置文件（chapter8/sql-map-config.xml）定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig PUBLIC "-//ibatis.apache.org//DTD SQL Map Config
2.0//EN"
      "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
      <settings enhancementEnabled="true" useStatementNamespaces="true"
            maxTransactions="20" maxRequests="32" maxSessions="10"/>
      <sqlMap resource="chapter8/sqlmaps/UserSQL.xml"/>
</sqlMapConfig>
```

5、数据源定义，此处使用第 7 章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

6、SqlMapClient 配置（chapter8/applicationContext-ibatis.xml）定义：

```
<bean id="sqlMapClient"
        class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <!-- 1、指定数据源 -->
    <property name="dataSource" ref="dataSource"/>
    <!-- 2、指定配置文件 -->
    <property name="configLocation" value="chapter8/sql-map-config.xml"/>
</bean>
```

7、获取 SqlMapClient：

```
package cn.javass.spring.chapter8;
//省略 import
public class IbatisTest {
    private static SqlMapClient sqlMapClient;
    @BeforeClass
    public static void setUpClass() {
        String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-ibatis.xml"};
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(configLocations);
        sqlMapClient = ctx.getBean(SqlMapClient.class);
    }
}
```

此处我们使用了 chapter7/applicationContext-resources.xml 定义的“dataSource”数据源，通过 ctx.getBean(SqlMapClient.class)获取 SqlMapClient。

8、通过 SqlMapClient 创建和删除表：

```
@Before
public void setUp() throws SQLException {
    sqlMapClient.update("UserSQL.createTable");
}
@After
public void tearDown() throws SQLException {
    sqlMapClient.update("UserSQL.dropTable");
}
```

9、 使用 SqlMapClient 进行对象持久化：

```
@Test
public void testFirst() throws SQLException {
	UserModel model = new UserModel();
	model.setMyName("test");
	SqlMapSession session = null;
	try {
		session = sqlMapClient.openSession();
		beginTransaction(session);
		session.insert("UserSQL.insert", model);
		commitTransaction(session);
	} catch (SQLException e) {
		rollbackTransacrion(session);
		throw e;
	} finally {
		closeSession(session);
	}
}
private void closeSession(SqlMapSession session) {
	session.close();
}
private void rollbackTransacrion(SqlMapSession session) throws SQLException {
	if(session != null) {
		session.endTransaction();
	}
}
private void commitTransaction(SqlMapSession session) throws SQLException {
	session.commitTransaction();
}
private void beginTransaction(SqlMapSession session) throws SQLException {
	session.startTransaction();
}
```

同样令人心烦的事务管理和冗长代码，Spring 通用提供了 SqlMapClientTemplate 模板类来解决这些问题。

8.3.2 使用 SqlMapClientTemplate

SqlMapClientTemplate 模板类同样用于简化事务管理及常见操作，类似于 JdbcTemplate 模板类，对于复杂操作通过提供 SqlMapClientCallback 回调接口来允许更复杂的操作。

接下来示例一下 SqlMapClientTemplate 的使用：

```
@Test
public void testSqlMapClientTemplate() {
    SqlMapClientTemplate sqlMapClientTemplate =
        new SqlMapClientTemplate(sqlMapClient);
    final UserModel model = new UserModel();
    model.setMyName("myName");
    sqlMapClientTemplate.insert("UserSQL.insert", model);
    //通过回调允许更复杂操作
    sqlMapClientTemplate.execute(new SqlMapClientCallback<Void>() {
        @Override
        public Void doInSqlMapClient(SqlMapExecutor session)
            throws SQLException {
            session.insert("UserSQL.insert", model);
            return null;
    }});
}
```

通过 new SqlMapClientTemplate(sqlMapClient)创建 HibernateTemplate 模板类对象，通过调用模板类的 save 方法持久化对象，并且自动享受到 Spring 管理事务的好处。

而且 SqlMapClientTemplate 提供使用 SqlMapClientCallback 回调接口的方法 execute 用来支持复杂操作，当然也自动享受到 Spring 管理事务的好处。

8.3.3 集成 iBATIS 及最佳实践

类似于 JdbcDaoSupport 类，Spring 对 iBATIS 也提供了 SqlMapClientDaoSupport 类来支持一致的数据库访问。SqlMapClientDaoSupport 也是 DaoSupport 实现：

接下来示例一下 Spring 集成 iBATIS 的最佳实践：

1、 定义 Dao 接口，此处使用 cn.javass.spring.chapter7.dao.IUserDao：

2、 定义 Dao 接口实现，此处是 iBATIS 实现：

```
package cn.javass.spring.chapter8.dao.ibatis;
//省略import
public class UserIbatisDaoImpl extends SqlMapClientDaoSupport
        implements IUserDao {
        @Override
        public void save(UserModel model) {
                getSqlMapClientTemplate().insert("UserSQL.insert", model);
        }
        @Override
        public int countAll() {
                return (Integer) getSqlMapClientTemplate().
                                        queryForObject("UserSQL.countAll");
        }
}
```

3、修改 iBATS 映射文件（chapter8/sqlmaps/UserSQL.xml），添加 countAll 查询：

```
<select id="countAll" resultClass="java.lang.Integer">
        <![CDATA[ select count(*) from test ]]>
</select>
```

此处注意首先 iBATIS 实现放在 dao.ibaitis 包里，其次实现类命名如 UserIbatisDaoImpl，即×××IbatisDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

4、进行资源配置，使用 resources/chapter7/applicationContext-resources.xml：

5、dao 定义配置，在 chapter8/applicationContext-ibatis.xml 中添加如下配置：

```
<bean id="abstractDao" abstract="true">
        <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
<bean id="userDao"
          class="cn.javass.spring.chapter8.dao.ibatis.UserIbatisDaoImpl"
          parent="abstractDao"/>
```

首先定义抽象的 abstractDao，其有一个 sqlMapClient 属性，从而可以让继承的子类自动继承 sqlMapClient 属性注入；然后定义 userDao，且继承 abstractDao，从而继承 sqlMapClient 注入；我们在此给配置文件命名为

applicationContext-ibatis.xml 表示 iBAITIS 实现。

6、 最后测试一下吧（cn.javass.spring.chapter8. IbatisTest）：

```
@Test
public void testBestPractice() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-ibatis.xml"};
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(configLocations);
        IUserDao userDao = ctx.getBean(IUserDao.class);
        UserModel model = new UserModel();
        model.setMyName("test");
        userDao.save(model);
        Assert.assertEquals(1, userDao.countAll());
}
```

和 Spring JDBC 框架的最佳实践完全一样，除了使用 applicationContext-ibatis.xml 代替了 applicationContext-jdbc.xml，其他完全一样。也就是说，DAO 层的实现替换可以透明化。

8.3.4 Spring+iBATIS 的 CURD

Spring 集成 iBATIS 进行 CURD（增删改查），也非常简单，首先配置映射文件，然后调用 SqlMapClientTemplate 相应的函数进行操作即可，此处就不介绍了。

8.3.5 集成 MyBatis 及最佳实践

2010 年 4 月份 iBATIS 团队发布 iBATIS 3.0 的 GA 版本的候选版本，在 iBATIS 3 中引入了泛型、注解支持等，因此需要 Java5+才能使用，但在 2010 年 6 月 16 日，iBATIS 团队决定从 apache 迁出并迁移到 Google Code，并更名为 MyBatis。目前新网站上文档并不完善。

目前 iBATIS 2.x 和 MyBatis 3 不是 100%兼容的，如配置文件的 DTD 变更，SqlMapClient 直接由 SqlSessionFactory 代替了，包名也有 com.ibatis 变成 org.ibatis 等等。

ibatis 3.x 和 MyBatis 是兼容的，只需要将 DTD 变更一下就可以了。

感兴趣的朋友可以到 http://www.mybatis.org/官网去下载最新的文档学习，作者只使用过 iBATIS 2.3.4 及以前版本，没在新项目使用过最新的 iBATIS 3.x 和 Mybatis，因此如果读者需要在项目中使用最新的 MyBatis，请先做好调研再使用。

接下来示例一下 Spring 集成 MyBatis 的最佳实践：

1、准备需要的 jar 包，到 MyBatis 官网下载 mybatis 3.0.4 版本和 mybatis-spring 1.0.0 版本，并拷贝如下 jar 包到类路径：

```
mybatis-3.0.4\mybatis-3.0.4.jar                    //核心 MyBatis 包
mybatis-spring-1.0.0\mybatis-spring-1.0.0.jar    //集成 Spring 包
```

2、对象模型定义，此处使用第七章中的 UserModel；

3、 MyBatis 映射定义（chapter8/sqlmaps/UserSQL-mybatis.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="UserSQL">
    <sql id="createTable">
    <!--id自增主键从0开始 -->
    <![CDATA[
       create memory table test(
          id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
          name varchar(100))
    ]]>
    </sql>
    <sql id="dropTable">
    <![CDATA[ drop table test ]]>
    </sql>
    <insert id="insert" parameterType="cn.javass.spring.chapter7.UserModel">
    <![CDATA[ insert into test(name) values (#{myName}) ]]>
          <selectKey resultType="int" keyProperty="id" order="AFTER">
              <!-- 获取hsqldb插入的主键 -->
              call identity();
              <!-- mysql使用select last_insert_id();获取插入的主键 -->
          </selectKey>
    </insert>
    <select id="countAll" resultType="java.lang.Integer">
    <![CDATA[ select count(*) from test ]]>
    </select>
</mapper>
```

从映射定义中可以看出 MyBatis 与 iBATIS 2.3.4 有如下不同：

- http://ibatis.apache.org/dtd/sql-map-2.dtd 废弃，而使用

http://mybatis.org/dtd/mybatis-3-mapper.dtd。

- <sqlMap>废弃，而使用<mapper>标签；
- <statement>废弃了，而使用<sql>标签；
- parameterClass 属性废弃，而使用 parameterType 属性；
- resultClass 属性废弃，而使用 resultType 属性；
- #myName#方式指定命名参数废弃，而使用#{myName}方式。

4、 MyBatis 配置文件（chapter8/sql-map-config-mybatis.xml）定义：

```
<?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>
    <settings>
        <setting name="cacheEnabled" value="false"/>
    </settings>
    <mappers>
        <mapper resource="chapter8/sqlmaps/UserSQL-mybatis.xml"/>
    </mappers>
</configuration>
```

从配置定义中可以看出 MyBatis 与 iBATIS 2.3.4 有如下不同：

- http://ibatis.apache.org/dtd/sql-map-config-2.dtd 废弃，而使用 http://mybatis.org/dtd/mybatis-3-config.dtd；
- < sqlMapConfig >废弃，而使用<configuration>；
- settings 属性配置方式废弃，而改用子标签< setting name=".." value=".."/>方式指定属性，且一些属性被废弃，如 maxTransactions；
- < sqlMap>废弃，而采用<mappers>标签及其子标签<mapper>定义。

5、 定义 Dao 接口，此处使用 cn.javass.spring.chapter7.dao. IUserDao；

6、 定义 Dao 接口实现，此处是 MyBatis 实现：

```
package cn.javass.spring.chapter8.dao.mybatis;
//省略 import
public class UserMybatisDaoImpl extends SqlSessionDaoSupport
    implements IUserDao {
    @Override
    public void save(UserModel model) {
        getSqlSession().insert("UserSQL.insert", model);
    }
    @Override
    public int countAll() {
        return (Integer) getSqlSession().selectOne("UserSQL.countAll");
    }
}
```

和 Ibatis 集成方式不同的有如下地方：

- 使用 SqlSessionDaoSupport 来支持一致性的 DAO 访问，该类位于 org.mybatis.spring.support 包中，非 Spring 提供；
- 使用 getSqlSession 方法获取 SqlSessionTemplate，在较早版本中是 getSqlSessionTemplate 方法名，不知为什么改成 getSqlSession 方法名，因此这个地方在使用时需要注意。
- SqlSessionTemplate 是 SqlSession 接口的实现，并且自动享受 Spring 管理事务好处，因此从此处可以推断出为什么把获取模板类的方法名改为 getSqlSession 而不是 getSqlSessionTemplate。

6、进行资源配置，使用 resources/chapter7/applicationContext-resources.xml：

7、dao 定义配置，在 chapter8/applicationContext-mybatis.xml 中添加如下配置：

```
<bean id="sqlSessionFactory"
        class="org.mybatis.spring.SqlSessionFactoryBean">
   <property name="dataSource" ref="dataSource"/><!-- 1、指定数据源 -->
   <property name="configLocation"
             value="chapter8/sql-map-config-mybatis.xml"/>
</bean>
<bean id="abstractDao" abstract="true">
      <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
<bean id="userDao"
       class="cn.javass.spring.chapter8.dao.mybatis.UserMybatisDaoImpl"
       parent="abstractDao"/>
```

和 Ibatis 集成方式不同的有如下地方：

- SqlMapClient 类废弃，而使用 SqlSessionFactory 代替；
- 使用 SqlSessionFactoryBean 进行集成 MyBatis。

首先定义抽象的 abstractDao，其有一个 sqlSessionFactory 属性，从而可以让继承的子类自动继承 sqlSessionFactory 属性注入；然后定义 userDao，且继承 abstractDao，从而继承 sqlSessionFactory 注入；我们在此给配置文件命名为 applicationContext-mybatis.xml 表示 MyBatis 实现。

8、最后测试一下吧（cn.javass.spring.chapter8. IbatisTest）：

```
@Test
public void testMybatisBestPractice() {
    String[] configLocations = new String[] {
        "classpath:chapter7/applicationContext-resources.xml",
        "classpath:chapter8/applicationContext-mybatis.xml"};
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和 Spring 集成 Ibatis 的最佳实践完全一样，除了使用 applicationContext-mybatis.xml 代替了 applicationContext-ibatis.xml，其他完全一样，且 MyBatis 3.x 与 Spring 整合只能运行在 Spring3.x。

在写本书时，MyBatis 与 Spring 集成所定义的 API 不稳定，且期待 Spring 能在发布新版本时将加入对 MyBatis 的支持。

8.4 集成 JPA

JPA 全称为 Java 持久性 API（Java Persistence API），JPA 是 Java EE 5 标准之一，是一个 ORM 规范，由厂商来实现该规范，目前有 Hibernate、OpenJPA、TopLink、EclipseJPA 等实现。

8.4.1 如何集成

Spring 目前提供集成 Hibernate、OpenJPA、TopLink、EclipseJPA 四个 JPA 标准实现。Spring 通过使用如下 Bean 进行集成 JPA（EntityManagerFactory）：

- **LocalEntityManagerFactoryBean：**适用于那些仅使用 JPA 进行数据访问的项目，该 FactoryBean 将根据 JPA PersistenceProvider 自动检测配置文件进行工作，一般从“META-INF/persistence.xml”读取配置信息，这种方式最简单，但不能设置 Spring 中定义的 DataSource，且不支持 Spring 管理的全局事务，

而且 JPA 实现商可能在 JVM 启动时依赖于 VM agent 从而允许它们进行持久化类字节码转换（不同的实现厂商要求不同，需要时阅读其文档），不建议使用这种方式；

- persistenceUnitName：指定持久化单元的名称；
- 使用方式：

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
      <property name="persistenceUnitName" value="persistenceUnit"/>
</bean>
```

- **从 JNDI 中获取：**用于从 Java EE 服务器获取指定的 EntityManagerFactory，这种方式在进行 Spring 事务管理时一般要使用 JTA 事务管理；
  - 使用方式：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">
<jee:jndi-lookup id="entityManagerFactory"
                 jndi-name="persistence/persistenceUnit"/>
</beans>
```

此处需要使用“jee”命名标签，且使用<jee:jndi-lookup>标签进行 JNDI 查找，“jndi-name”属性用于指定 JNDI 名字。

- **LocalContainerEntityManagerFactoryBean:**适用于所有环境的 FactoryBean，能全面控制 EntityManagerFactory 配置,如指定 Spring 定义的 DataSource 等等。
  - **persistenceUnitManager**：用于获取 JPA 持久化单元，默认实现 DefaultPersistenceUnitManager 用于解决多配置文件情况
  - **dataSource**：用于指定 Spring 定义的数据源；
  - **persistenceXmlLocation**：用于指定 JPA 配置文件，对于对配置文件情况请选择设置 persistenceUnitManager 属性来解决；
  - **persistenceUnitName**：用于指定持久化单元名字；
  - **persistenceProvider**：用于指定持久化实现厂商类；如 Hibernate 为 org.hibernate.ejb.HibernatePersistence 类；
  - **jpaVendorAdapter**：用于设置实现厂商 JPA 实现的特定属性，如设置 Hibernate 的是否自动生成 DDL 的属性 generateDdl;这些属性是厂商特定的，

因此最好在这里设置；目前 Spring 提供 HibernateJpaVendorAdapter、OpenJpaVendorAdapter 、EclipseLinkJpaVendorAdapter 、TopLinkJpaVendorAdapter、OpenJpaVendorAdapter 四个实现。其中最重要的属性是“**database**”，用来指定使用的数据库类型，从而能根据数据库类型来决定比如如何将数据库特定异常转换为 Spring 的一致性异常，目前支持如下数据库（**DB2、DERBY、H2、HSQL、INFORMIX、MYSQL、ORACLE、POSTGRESQL、SQL_SERVER、SYBASE**）。

- **jpaDialect**：用于指定一些高级特性，如事务管理，获取具有事务功能的连接对象等，目前 Spring 提供 HibernateJpaDialect、OpenJpaDialect 、EclipseLinkJpaDialect、TopLinkJpaDialect、和 DefaultJpaDialect 实现，注意 DefaultJpaDialect 不提供任何功能，因此在使用特定实现厂商 JPA 实现时需要指定 JpaDialect 实现，如使用 Hibernate 就使用 HibernateJpaDialect。当指定 **jpaVendorAdapter 属性时可以不指定 jpaDialect，会自动设置相应的 JpaDialect 实现；**
- **jpaProperties 和 jpaPropertyMap**：指定 JPA 属性；如 Hibernate 中指定是否显示 SQL 的“hibernate.show_sql”属性，对于 jpaProperties 设置的属性自动会放进 jpaPropertyMap 中；
- loadTimeWeaver：用于指定 LoadTimeWeaver 实现，从而允许 JPA 加载时修改相应的类文件。具体使用得参考相应的 JPA 规范实现厂商文档，如 Hibernate 就不需要指定 loadTimeWeaver。

接下来学习一下 Spring 如何集成 JPA 吧：

1、准备 jar 包，从下载的 hibernate-distribution-3.6.0.Final 包中获取如下 Hibernate 需要的 jar 包从而支持 JPA：

```
lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar //用于支持 JPA
```

2、对象模型定义，此处使用 UserModel2：

```
package cn.javass.spring.chapter8;
//省略 import
@Entity
@Table(name = "test")
public class UserModel2 {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(name = "name")
    private String myName;
    //省略 getter 和 setter
}
```

用@org.hibernate.annotations.Entity 而非@javax.persistence.Entity 将导致 JPA 不能正常工作。

3、 JPA 配置定义（chapter8/persistence.xml），定义对象和数据库之间的映射：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="persistenceUnit"
                transaction-type="RESOURCE_LOCAL"/>
</persistence>
```

在 JPA 配置文件中，我们指定要持久化单元名字，和事务类型，其他都将在 Spring 中配置。

4、数据源定义，此处使用第 7 章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

5、EntityManagerFactory 配置定义（chapter8/applicationContext-jpa.xml）：

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.
            LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="persistenceXmlLocation"
            value="chapter8/persistence.xml"/>
    <property name="persistenceUnitName" value="persistenceUnit"/>
    <property name="persistenceProvider" ref="persistenceProvider"/>
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
    <property name="jpaDialect" ref="jpaDialect"/>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>
<bean id="persistenceProvider"
     class="org.hibernate.ejb.HibernatePersistence"/>
```

```
<bean id="jpaVendorAdapter"
    class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="generateDdl" value="false" />
    <property name="database" value="HSQL"/>
</bean>
<bean id="jpaDialect"
```

- **LocalContainerEntityManagerFactoryBean:** 指定使用本地容器管理EntityManagerFactory，从而进行细粒度控制；
- **dataSource** 属性指定使用 Spring 定义的数据源；
- **persistenceXmlLocation** 指定 JPA 配置文件为 chapter8/persistence.xml，且该配置文件非常简单，具体配置完全在 Spring 中进行；
- **persistenceUnitName** 指定持久化单元名字，即 JPA 配置文件中指定的;
- **persistenceProvider**:指定 JPA 持久化提供商，此处使用 Hibernate 实现 HibernatePersistence 类；
- **jpaVendorAdapter**：指定实现厂商专用特性，即 generateDdl= false 表示不自动生成 DDL，database= HSQL 表示使用 hsqldb 数据库；
- **jpaDialect**：如果指定 jpaVendorAdapter 此属性可选，此处为 HibernateJpaDialect；
- **jpaProperties**：此处指定“hibernate.show_sql =true”表示在日志系统 debug 级别下将打印所有生成的 SQL。

6、 获取 EntityManagerFactory：

```
package cn.javass.spring.chapter8;
//省略 import
public class JPATest {
    private static EntityManagerFactory entityManagerFactory;
    @BeforeClass
    public static void setUpClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-jpa.xml"};
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(configLocations);
        entityManagerFactory = ctx.getBean(EntityManagerFactory.class);
    }
}
```

此处我们使用了 chapter7/applicationContext-resources.xml 定义的“dataSource”数据源，通过 ctx.getBean(EntityManagerFactory.class)获取 EntityManagerFactory。

7、 通过 EntityManagerFactory 获取 EntityManager 进行创建和删除表：

```
@Before
public void setUp() throws SQLException {
   //id自增主键从0开始
   String createTableSql = "create memory table test" +
      "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
      "name varchar(100))";
   executeSql(createTableSql);
}
@After
public void tearDown() throws SQLException {
   String dropTableSql = "drop table test";
   executeSql(dropTableSql);
}
```

```
private void executeSql(String sql) throws SQLException {
   EntityManager em = entityManagerFactory.createEntityManager();
   beginTransaction(em);
   em.createNativeQuery(sql).executeUpdate();
   commitTransaction(em);
   closeEntityManager(em);
}
private void closeEntityManager(EntityManager em) {
   em.close();
}
private void rollbackTransacrion(EntityManager em) throws SQLException {
   if(em != null) {
      em.getTransaction().rollback();
   }
}
private void commitTransaction(EntityManager em) throws SQLException {
   em.getTransaction().commit();
}
private void beginTransaction(EntityManager em) throws SQLException {
   em.getTransaction().begin();
}
```

使用 EntityManagerFactory 创建 EntityManager，然后通过 EntityManager 对象的 createNativeQuery 创建本地 SQL 执行创建和删除表。

8、使用 EntityManagerFactory 获取 EntityManager 对象进行持久化数据：

```
@Test
public void testFirst() throws SQLException {
    UserModel2 model = new UserModel2();
    model.setMyName("test");
    EntityManager em = null;
    try {
        em = entityManagerFactory.createEntityManager();
        beginTransaction(em);
        em.persist(model);
        commitTransaction(em);
    } catch (SQLException e) {
        rollbackTransacrion(em);
        throw e;
    } finally {
       closeEntityManager(em);
    }
}
```

使用 **EntityManagerFactory** 获取 **EntityManager** 进行操作，看到这还能忍受冗长的代码和事务管理吗？Spring 同样提供 JpaTemplate 模板类来简化这些操作。

大家有没有注意到此处的模型对象能自动映射到数据库，这是因为 Hibernate JPA 实现默认自动扫描类路径中的@Entity 注解类及*.hbm.xml 映射文件，可以通过更改 Hibernate JPA 属性“hibernate.ejb.resource_scanner”，并指定 org.hibernate.ejb.packaging.Scanner 接口实现来定制新的扫描策略。

8.4.2 使用 JpaTemplate

JpaTemplate 模板类用于简化事务管理及常见操作，类似于 JdbcTemplate 模板类，对于复杂操作通过提供 JpaCallback 回调接口来允许更复杂的操作。

接下来示例一下 JpaTemplate 的使用：

1、修改 Spring 配置文件（chapter8/applicationContext-jpa.xml），添加 JPA 事务管理器：

```
<bean id="txManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

- txManager：指定事务管理器，JPA 使用 JpaTransactionManager 事务管理器实现，通过 entityManagerFactory 指定 EntityManagerFactory；用于支持 Java SE 环境的 **JPA 扩展的持久化上下文（EXTENDED Persistence Context）**。

2、修改 JPATest 类，添加类变量 ctx，用于后边使用其获取事务管理器使用：

```
package cn.javass.spring.chapter8;
public class JPATest {
    private static EntityManagerFactory entityManagerFactory;
    private static ApplicationContext ctx;
    @BeforeClass
    public static void beforeClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-jpa.xml"};
        ctx = new ClassPathXmlApplicationContext(configLocations);
        entityManagerFactory = ctx.getBean(EntityManagerFactory.class);
    }
}
```

3）JpaTemplate 模板类使用：

```
@Test
public void testJpaTemplate() {
   final JpaTemplate jpaTemplate = new JpaTemplate(entityManagerFactory);
   final UserModel2 model = new UserModel2();
   model.setMyName("test1");
   PlatformTransactionManager txManager =
      ctx.getBean(PlatformTransactionManager.class);
   new TransactionTemplate(txManager).execute(
      new TransactionCallback<Void>() {
         @Override
         public Void doInTransaction(TransactionStatus status) {
            jpaTemplate.persist(model);
            return null;
         }
   });
   String COUNT_ALL = "select count(*) from UserModel";
   Number count = (Number) jpaTemplate.find(COUNT_ALL).get(0);
   Assert.assertEquals(1, count.intValue());
}
```

- **jpaTemplate**：可通过 new JpaTemplate(entityManagerFactory)方式创建；
- **txManager**：通过 ctx.getBean(PlatformTransactionManager.class)获取事务管理器；
- **TransactionTemplate**：通过 new TransactionTemplate(txManager)创建事务模板对象，并通过 execute 方法执行 TransactionCallback 回调中的 doInTransaction 方法中定义需要执行的操作，从而将由模板类通过 txManager 事务管理器来进行事务管理，此处是调用 jpaTemplate 对象的 persist 方法进行持久化；
- **jpaTemplate.persist()**：根据 JPA 规范，在 JPA 扩展的持久化上下文，该操作必须运行在事务环境，还有 persist()、 merge()、remove()操作也必须运行在事务环境；
- **jpaTemplate.find()**：根据 JPA 规范，该操作无需运行在事务环境，还有 find()、getReference()、 refresh()、detach()和查询操作都无需运行在事务环境。

此实例与 Hibernate 和 Ibatis 有所区别，通过 JpaTemplate 模板类进行如持久化等操作时必须有运行在事务环境中，否则可能抛出如下异常或警告：

- **“ javax.persistence.TransactionRequiredException : Executing an update/delete query”**：表示没有事务支持，不能执行更新或删除操作；
- **警告“delaying identity-insert due to no transaction in progress”**：需要在日志系统启动 debug 模式才能看到，表示在无事务环境中无法进行持久化，而选择了延迟标识插入。

以上异常和警告是没有事务造成的，也是最让人困惑的问题，需要大家注意。

8.4.3 集成 JPA 及最佳实践

类似于 JdbcDaoSupport 类，Spring 对 JPA 也提供了 JpaDaoSupport 类来支持一致的数据库访问。JpaDaoSupport 也是 DaoSupport 实现：

接下来示例一下 Spring 集成 JPA 的最佳实践：

1、 定义 Dao 接口，此处使用 cn.javass.spring.chapter7.dao. IUserDao：

2、 定义 Dao 接口实现，此处是 JPA 实现：

```
package cn.javass.spring.chapter8.dao.jpa;
//省略 import
@Transactional(propagation = Propagation.REQUIRED)
public class UserJpaDaoImpl extends JpaDaoSupport implements IUserDao {
    private static final String COUNT_ALL_JPAQL =
        "select count(*) from UserModel";
    @Override
    public void save(UserModel model) {
        getJpaTemplate().persist(model);
    }
    @Override
    public int countAll() {
        Number count =
            (Number) getJpaTemplate().find(COUNT_ALL_JPAQL).get(0);
        return count.intValue();
    }
}
```

此处注意首先 JPA 实现放在 dao.jpa 包里，其次实现类命名如 UserJpaDaoImpl，即×××JpaDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

另外在类上添加了**@Transactional** 注解表示该类的所有方法将在调用时需要事务支持，propagation 传播属性为 Propagation.REQUIRED 表示事务是必需的，如果执行该类的方法没有开启事务，将开启一个新的事务。

3、进行资源配置，使用 resources/chapter7/applicationContext-resources.xml：

4、dao 定义配置，在 chapter8/applicationContext-jpa.xml 中添加如下配置：

4.1、首先添加 tx 命名空间用于支持事务：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

4.2、为**@Transactional** 注解事务开启事务支持：

```
<tx:annotation-driven transaction-manager="txManager"/>
```

只为类添加**@Transactional** 注解是不能支持事务的，需要通过<tx:annotation-driven>标签来开启事务支持，其中 txManager 属性指定事务管理器。

4.3、配置 DAO Bean：

```
<bean id="abstractDao" abstract="true">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
<bean id="userDao"
        class="cn.javass.spring.chapter8.dao.jpa.UserJpaDaoImpl"
        parent="abstractDao"/>
```

首先定义抽象的 abstractDao，其有一个 entityManagerFactory 属性，从而可以让继承的子类自动继承 entityManagerFactory 属性注入；然后定义 userDao，且继承 abstractDao，从而继承 entityManagerFactory 注入；我们在此给配置文件命名为 applicationContext-jpa.xml 表示 JPA 实现。

- **最后测试一下吧（cn.javass.spring.chapter8. JPATest）：**

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-jpa.xml"};
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和 Spring JDBC 框架的最佳实践完全一样，除了使用 applicationContext-jpa.xml 代替了 applicationContext-jdbc.xml，其他完全一样。也就是说，DAO 层的实现替换可以透明化。

还有与集成其他 ORM 框架不同的是 JPA 在进行持久化或更新数据库操作时

需要事务支持。

8.4.4 Spring+JPA 的 CRUD

Spring+JPA CRUD（增删改查）也相当简单，让我们直接看具体示例吧：

```
@Test
public void testCRUD() {
    PlatformTransactionManager txManager =
        ctx.getBean(PlatformTransactionManager.class);
    final JpaTemplate jpaTemplate = new JpaTemplate(entityManagerFactory);
    TransactionTemplate tansactionTemplate =
        new TransactionTemplate(txManager);
    tansactionTemplate.execute(new TransactionCallback<Void>() {
        @Override
        public Void doInTransaction(TransactionStatus status) {
            UserModel model = new UserModel();
            model.setMyName("test");
            //新增
            jpaTemplate.persist(model);
            //修改
            model.setMyName("test2");
            jpaTemplate.flush();//可选
            //查询
            String sql = "from UserModel where myName=?";
            List result = jpaTemplate.find(sql, "test2");
            Assert.assertEquals(1, result.size());
            //删除
            jpaTemplate.remove(model);
            return null;
        }
    });
}
```

- 对于增删改必须运行在事务环境，因此我们使用 TransactionTemplate 事务模板类来支持事务。
- 持久化：使用 JpaTemplate 类的 persist 方法持久化模型对象；
- 更新：对于持久化状态的模型对象直接修改属性，调用 flush 方法即可更新到数据库，在一些场合时 flush 方法调用可选，如执行一个查询操作等，具体请

参考相关文档；

- 查询：可以使用 find 方法执行 JPA QL 查询；
- 删除：使用 remove 方法删除一个持久化状态的模型对象。

Spring 集成 JPA 进行增删改查也相当简单，但本文介绍的稍微复杂一点，因为牵扯到编程式事务，如果采用声明式事务将和集成 Hibernate 方式一样简洁。

第九章　Spring 的事务

9.1　数据库事务概述

事务首先是一系列操作组成的工作单元，该工作单元内的操作是不可分割的，即要么所有操作都做，要么所有操作都不做，这就是事务。

事务必需满足 ACID（原子性、一致性、隔离性和持久性）特性，缺一不可：

- **原子性（Atomicity）**：即事务是不可分割的最小工作单元，事务内的操作要么全做，要么全不做；
- **一致性（Consistency）**：在事务执行前数据库的数据处于正确的状态，而事务执行完成后数据库的数据还是处于正确的状态，即数据完整性约束没有被破坏；如银行转帐，A 转帐给 B，必须保证 A 的钱一定转给 B，一定不会出现 A 的钱转了但 B 没收到，否则数据库的数据就处于不一致（不正确）的状态。
- **隔离性（Isolation）**：并发事务执行之间无影响，在一个事务内部的操作对其他事务是不产生影响，这需要事务隔离级别来指定隔离性；
- **持久性（Durability）**：事务一旦执行成功，它对数据库的数据的改变必须是永久的，不会因比如遇到系统故障或断电造成数据不一致或丢失。

在实际项目开发中数据库操作一般都是并发执行的，即有多个事务并发执行，并发执行就可能遇到问题，目前常见的问题如下：

- 丢失更新：两个事务同时更新一行数据，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失，这是由于没有加锁造成的；
- 脏读：一个事务看到了另一个事务未提交的更新数据；
- 不可重复读：在同一事务中，多次读取同一数据却返回不同的结果；也就是有其他事务更改了这些数据；
- 幻读：一个事务在执行过程中读取到了另一个事务已提交的插入数据；即在第一个事务开始时读取到一批数据，但此后另一个事务又插入了新数据并提

交，此时第一个事务又读取这批数据但发现多了一条，即好像发生幻觉一样。

为了解决这些并发问题，需要通过数据库隔离级别来解决，在标准 SQL 规范中定义了四种隔离级别：

- **未提交读（Read Uncommitted）**：最低隔离级别，一个事务能读取到别的事务未提交的更新数据，很不安全，可能出现丢失更新、脏读、不可重复读、幻读；
- **提交读（Read Committed）**：一个事务能读取到别的事务提交的更新数据，不能看到未提交的更新数据，不可能可能出现丢失更新、脏读，但可能出现不可重复读、幻读；
- **可重复读（Repeatable Read）**：保证同一事务中先后执行的多次查询将返回同一结果，不受其他事务影响，可能可能出现丢失更新、脏读、不可重复读，但可能出现幻读；
- **序列化（Serializable）**：最高隔离级别，不允许事务并发执行，而必须串行化执行，最安全，不可能出现更新、脏读、不可重复读、幻读。

隔离级别越高，数据库事务并发执行性能越差，能处理的操作越少。因此在实际项目开发中为了考虑并发性能一般使用**提交读**隔离级别，它能避免丢失更新和脏读，尽管不可重复读和幻读不能避免，但可以在可能出现的场合使用**悲观锁或乐观锁**来解决这些问题。

9.1.1 事务类型

数据库事务类型有本地事务和分布式事务：

- 本地事务：就是普通事务，能保证单台数据库上的操作的 ACID，被限定在一台数据库上；
- 分布式事务：涉及两个或多个数据库源的事务，即跨越多台同类或异类数据库的事务（由每台数据库的本地事务组成的），分布式事务旨在保证这些本地事务的所有操作的 ACID，使事务可以跨越多台数据库；

Java 事务类型有 JDBC 事务和 JTA 事务：

- JDBC 事务：就是数据库事务类型中的本地事务，通过 Connection 对象的控制来管理事务；
- JTA 事务：JTA 指 Java 事务 API(Java Transaction API)，是 Java EE 数据库事务规范，JTA 只提供了事务管理接口，由应用程序服务器厂商（如 WebSphere Application Server）提供实现，JTA 事务比 JDBC 更强大，支持分布式事务。

Java EE 事务类型有本地事务和全局事务：

- 本地事务：使用 JDBC 编程实现事务；
- 全局事务：由应用程序服务器提供，使用 JTA 事务；

按是否通过编程实现事务有声明式事务和编程式事务；

- 声明式事务： 通过注解或 XML 配置文件指定事务信息；
- 编程式事务：通过编写代码实现事务。

9.1.2 Spring 提供的事务管理

Spring 框架最核心功能之一就是事务管理，而且提供一致的事务管理抽象，这能帮助我们：

- 提供一致的编程式事务管理 API，不管使用 Spring JDBC 框架还是集成第三方框架使用该 API 进行事务编程；
- 无侵入式的声明式事务支持。

Spring 支持声明式事务和编程式事务事务类型。

9.2 事务管理器

9.2.1 概述

Spring 框架支持事务管理的核心是事务管理器抽象，对于不同的数据访问框架（如 Hibernate）通过实现策略接口 PlatformTransactionManager，从而能支持各种数据访问框架的事务管理，PlatformTransactionManager 接口定义如下：

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

- **getTransaction()**：返回一个已经激活的事务或创建一个新的事务（根据给定的 TransactionDefinition 类型参数定义的事务属性），返回的是 TransactionStatus 对象代表了当前事务的状态，其中该方法抛出 TransactionException（未检查异常）表示事务由于某种原因失败。
- **commit()**：用于提交 TransactionStatus 参数代表的事务，具体语义请参考 Spring Javadoc；
- **rollback()**：用于回滚 TransactionStatus 参数代表的事务，具体语义请参考 Spring Javadoc。

TransactionDefinition 接口定义如下：

```
public interface TransactionDefinition {
    int getPropagationBehavior();
    int getIsolationLevel();
    int getTimeout();
    boolean isReadOnly();
    String getName();
}
```

- **getPropagationBehavior()**：返回定义的事务传播行为；
- **getIsolationLevel()**：返回定义的事务隔离级别；
- **getTimeout()**：返回定义的事务超时时间；
- **isReadOnly()**：返回定义的事务是否是只读的；
- **getName()**：返回定义的事务名字。

TransactionStatus 接口定义如下：

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    void flush();
    boolean isCompleted();
}
```

- **isNewTransaction()：返回**当前事务状态是否是新事务**；**
- **hasSavepoint()：返回当前事务是否有保存点**；
- **setRollbackOnly()**：设置当前事务应该回滚；
- **isRollbackOnly(()**：返回当前事务是否应该回滚；
- **flush()：**用于刷新底层会话中的修改到数据库，一般用于刷新如 Hibernate/JPA 的会话，可能对如 JDBC 类型的事务无任何影响；
- **isCompleted():**当前事务否已经完成。

9.2.2 内置事务管理器实现

Spring 提供了许多内置事务管理器实现：

- **DataSourceTransactionManager**：位于 org.springframework.jdbc.datasource 包中，数据源事务管理器，提供对单个 javax.sql.DataSource 事务管理，用于 Spring JDBC 抽象框架、iBATIS 或 MyBatis 框架的事务管理；
- **JdoTransactionManager**：位于 org.springframework.orm.jdo 包中，提供对单个 javax.jdo.PersistenceManagerFactory 事务管理，用于集成 JDO 框架时的事务管理；
- **JpaTransactionManager**：位于 org.springframework.orm.jpa 包中，提供对单个 javax.persistence.EntityManagerFactory 事务支持，用于集成 JPA 实现框架时的事务管理；
- **HibernateTransactionManager**：位于 org.springframework.orm.hibernate3 包中，提供对单个 org.hibernate.SessionFactory 事务支持，用于集成 Hibernate

框架时的事务管理；该事务管理器只支持 Hibernate3+版本，且 Spring3.0+版本只支持 Hibernate 3.2+版本；

- **JtaTransactionManager：**位于 org.springframework.transaction.jta 包中，提供对分布式事务管理的支持，并将事务管理委托给 Java EE 应用服务器事务管理器；
- **OC4JjtaTransactionManager：**位于 org.springframework.transaction.jta 包中，Spring 提供的对 OC4J 10.1.3+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持；
- **WebSphereUowTransactionManager：**位于 org.springframework.transaction.jta 包中，Spring 提供的对 WebSphere 6.0+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持；
- **WebLogicJtaTransactionManager：**位于 org.springframework.transaction.jta 包中，Spring 提供的对 WebLogic 8.1+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持。

Spring 不仅提供这些事务管理器，还提供对如 JMS 事务管理的管理器等，Spring 提供一致的事务抽象如图 9-1 所示。

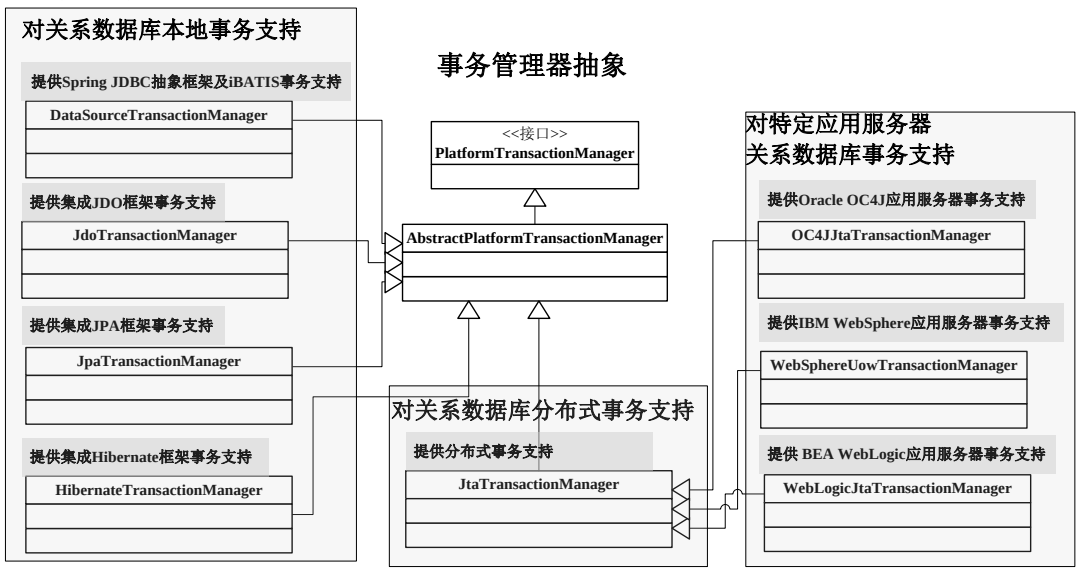

图 9-1 Spring 事务管理器

接下来让我们学习一下如何在 Spring 配置文件中定义事务管理器：

一、 声明对本地事务的支持：

a) JDBC 及 iBATIS、MyBatis 框架事务管理器

```
<bean id="txManager" class="
      org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource"/>
</bean>
```

通过 dataSource 属性指定需要事务管理的单个 javax.sql.DataSource 对象。

b) Jdo 事务管理器

```
<bean id="txManager" class="
      org.springframework.orm.jdo.JdoTransactionManager">
  <property name="persistenceManagerFactory"
            ref="persistenceManagerFactory"/>
</bean>
```

通过 persistenceManagerFactory 属性指定需要事务管理的 javax.jdo.PersistenceManagerFactory 对象。

c) Jpa 事务管理器

```
<bean id="txManager" class="
       org.springframework.orm.jpa.JpaTransactionManager">
   <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

通过 entityManagerFactory 属性指定需要事务管理的 javax.persistence.EntityManagerFactory 对象。

还需要为 entityManagerFactory 对象指定 jpaDialect 属性，该属性所对应的对象指定了如何获取连接对象、开启事务、关闭事务等事务管理相关的行为。

```
<bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.
               LocalContainerEntityManagerFactoryBean">
           ……
           <property name="jpaDialect" ref="jpaDialect"/>
</bean>
<bean id="jpaDialect"
      class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
```

d) Hibernate 事务管理器

```
<bean id="txManager" class="
      org.springframework.orm.hibernate3.HibernateTransactionManager">
   <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

通过 entityManagerFactory 属性指定需要事务管理的 org.hibernate.SessionFactory 对象。

二、 Spring 对全局事务的支持：

a) Jta 事务管理器

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

    <jee:jndi-lookup id="dataSource"
                jndi-name="jdbc/test"/>
    <bean id="txManager" class="
            org.springframework.transaction.jta.JtaTransactionManager">
        <property name="transactionManagerName"
                value=" java:comp/TransactionManager"/>
    </bean>
</beans>
```

“dataSource”Bean 表示从 JNDI 中获取的数据源，而 txManager 是 JTA 事务管理器，其中属性 transactionManagerName 指定了 JTA 事务管理器的 JNDI 名字，从而将事务管理委托给该事务管理器。

这只是最简单的配置方式，更复杂的形式请参考 Spring Javadoc。

在此我们再介绍两个不依赖于应用服务器的开源 JTA 事务实现：JOTM 和 Atomikos Transactions Essentials。

- **JOTM：**即基于 Java 开放事务管理器（Java Open Transaction Manager），实现 JTA 规范，能够运行在非应用服务器环境中，Web 容器或独立 Java SE 环境，官网地址: http://jotm.objectweb.org/。
- **Atomikos Transactions Essentials：**其为 Atomikos 开发的事务管理器，该产品属于开源产品，另外还一个商业的 Extreme Transactions。官网地址为：http://www.atomikos.com。

对于以上 JTA 事务管理器使用，本文作者只是做演示使用，如果在实际项目中需要不依赖于应用服务器的 JTA 事务支持，需详细测试并选择合适的。

在本文中将使用 Atomikos Transactions Essentials 来进行演示 JTA 事务使用，由于 Atomikos 对 hsqldb 分布式支持不是很好，在 Atomikos 官网中列出如下兼容的数据库：Oracle、Informix、FirstSQL、DB2、MySQL、SQLServer、Sybase，这不代表其他数据库不支持，而是 Atomikos 团队没完全测试，在此作者决定使用 derby 内存数据

库来演示 JTA 分布式事务。

1、首先准备 jar 包：

1.1 、 准 备 derby 数 据 jar 包 ， 到 下 载 的 spring-framework-3.0.5.RELEASE-dependencies.zip 中拷贝如下 jar 包：

```
com.springsource.org.apache.derby-10.5.1000001.764942.jar
```

1．2、准备 Atomikos Transactions Essentials 对 JTA 事务支持的 JTA 包，此处使用 AtomikosTransactionsEssentials 3.5.5 版本，到官网下载 AtomikosTransactionsEssentials-3.5.5.zip，拷贝如下 jar 包到类路径：

```
atomikos-util.jar
transactions-api.jar
transactions-essentials-all.jar
transactions-jdbc.jar
transactions-jta.jar
transactions.jar
```

将如上 jar 包放在 lib\atomikos 目录下，并添加到类路径中。

2、接下来看一下在 Spring 中如何配置 AtomikosTransactionsEssentials JTA 事务：

2.1、配置分布式数据源：

```
<bean id="dataSource1" class="com.atomikos.jdbc.AtomikosDataSourceBean"
                init-method="init" destroy-method="close">
    <property name="uniqueResourceName" value="jdbc/test1"/>
    <property name="xaDataSourceClassName"
                value="org.apache.derby.jdbc.EmbeddedXADataSource"/>
    <property name="poolSize" value="5"/>
    <property name="xaProperties">
        <props>
            <prop key="databaseName">test1</prop>
            <prop key="createDatabase">create</prop>
        </props>
    </property>
</bean>

<bean id="dataSource2" class="com.atomikos.jdbc.AtomikosDataSourceBean"
        init-method="init" destroy-method="close">
    ……
</bean>
```

在此我们配置两个分布式数据源：使用 com.atomikos.jdbc.AtomikosDataSourceBean 来配置 AtomikosTransactionsEssentials 分布式数据源：

- uniqueResourceName 表示唯一资源名，如有多个数据源不可重复；
- xaDataSourceClassName 是具体分布式数据源厂商实现；
- poolSize 是数据连接池大小；
- xaProperties 属性指定具体厂商数据库属性，如 databaseName 指定数据库名，createDatabase 表示启动 derby 内嵌数据库时创建 databaseName 指定的数据库。

在此我们还有定义了一个“dataSource2”Bean，其属性和“DataSource1”除以下不一样其他完全一样：

- uniqueResourceName：因为不能重复，因此此处使用 jdbc/test2；
- databaseName：我们需要指定两个数据库，因此此处我们指定为 test2。

2.2、配置事务管理器：

```
<bean id="atomikosTransactionManager"
        class = "com.atomikos.icatch.jta.UserTransactionManager"
        init-method="init" destroy-method = "close">
        <property name="forceShutdown" value="true"/>
</bean>
<bean id="atomikosUserTransaction"
       class="com.atomikos.icatch.jta.UserTransactionImp">
</bean>

<bean id="transactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager">
        <ref bean="atomikosTransactionManager"/>
    </property>
    <property name="userTransaction">
        <ref bean="atomikosUserTransaction"/>
    </property>
</bean>
```

- atomikosTransactionManager：定义了 AtomikosTransactionsEssentials 事务管理器；
- atomikosUserTransaction：定义 UserTransaction，该 Bean 是线程安全的；
- transactionManager：定义 Spring 事务管理器，transactionManager 属性指定外部事务管理器（真正的事务管理者），使用 userTransaction 指定

UserTransaction，该属性一般用于本地 JTA 实现，如果使用应用服务器事务管理器，该属性将自动从 JNDI 获取。

配置完毕，是不是也挺简单的，但是如果确实需要使用 JTA 事务，请首先选择应用服务器事务管理器，本示例不适合生产环境，如果非要运用到生产环境，可以考虑购买 AtomikosTransactionsEssentials 商业支持。

b) 特定服务器事务管理器

Spring 还提供了对特定应用服务器事务管理器集成的支持，目前提供对 IBM WebSphere、BEA WebLogic、 Oracle OC4J 应用服务器高级事务的支持，具体使用请参考 Spring Javadoc。

现在我们已经学习如何配置事务管理器了，但是只有事务管理器 Spring 能自动进行事务管理吗？当然不能了，这需要我们来控制，目前 Spring 支持两种事务管理方式：编程式和声明式事务管理。接下来先看一下如何进行编程式事务管理吧。

9.3 编程式事务

9.3.1 编程式事务概述

所谓编程式事务指的是通过编码方式实现事务，即类似于 JDBC 编程实现事务管理。

Spring 框架提供一致的事务抽象，因此对于 JDBC 还是 JTA 事务都是采用相同的 API 进行编程。

```
Connection conn = null;
UserTransaction tx = null;
try {
   tx = getUserTransaction();                          //1.获取事务
   tx.begin();                                         //2.开启JTA事务
   conn = getDataSource().getConnection();             //3.获取JDBC
                                                       //4.声明SQL
   String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
   PreparedStatement pstmt = conn.prepareStatement(sql);//5.预编译SQL
   ResultSet rs = pstmt.executeQuery();                //6.执行SQL
   process(rs);                                        //7.处理结果集
   closeResultSet(rs);                                 //8.释放结果集
   tx.commit();                                        //7.提交事务
} catch (Exception e) {
   tx.rollback();                                     //8.回滚事务
   throw e;
} finally {
   conn.close();                                     //关闭连接
}
```

此处可以看到使用 UserTransaction 而不是 Connection 连接进行控制事务，从而对于 JDBC 事务和 JTA 事务是采用不同 API 进行编程控制的，并且 JTA 和 JDBC 事务管理的异常也是不一样的。

具体如何使用 JTA 编程进行事务管理请参考 cn.javass.spring.chapter9 包下的 TranditionalTransactionTest 类。

而在 Spring 中将采用一致的事务抽象进行控制和一致的异常控制，即面向 PlatformTransactionManager 接口编程来控制事务。

9.3.1 Spring 对编程式事务的支持

Spring 中的事务分为物理事务和逻辑事务；

- **物理事务**：就是底层数据库提供的事务支持，如 JDBC 或 JTA 提供的事务；
- **逻辑事务**：是 Spring 管理的事务，不同于物理事务，逻辑事务提供更丰富的控制，而且如果想得到 Spring 事务管理的好处，必须使用逻辑事务，因此在 Spring 中如果没特别强调一般就是逻辑事务；

逻辑事务即支持非常低级别的控制，也有高级别解决方案：

- **低级别解决方案：**
  - **工具类：**使用工具类获取连接（会话）和释放连接（会话），如使用 org.springframework.jdbc.datasource 包中的 ConnectionUtils 类来获取和释放具有逻辑事务功能的连接。当然对集成第三方 ORM 框架也提供了类似的工具类，如对 Hibernate 提供了 SessionFactoryUtils 工具类，JPA 的 EntityManagerFactoryUtils 等，其他工具类都是使用类似***Utils 命名；

```
//获取具有 Spring 事务（逻辑事务）管理功能的连接
DataSourceUtils. getConnection(DataSource dataSource)
//释放具有 Spring 事务（逻辑事务）管理功能的连接
DataSourceUtils. releaseConnection(Connection con, DataSource
```

  - **TransactionAwareDataSourceProxy：**使用该数据源代理类包装需要 Spring 事务管理支持的数据源，该包装类必须位于最外层，主要用于遗留项目中可能直接使用数据源获取连接和释放连接支持或希望在 Spring 中进行混合使用各种持久化框架时使用，其内部实际使用 ConnectionUtils 工具类获取和释放真正连接；

```
<!--使用该方式包装数据源，必须在最外层，targetDataSource 知道目标
    数据源-->
<bean id="dataSourceProxy"
      class="org.springframework.jdbc.datasource.
      TransactionAwareDataSourceProxy">
    <property name="targetDataSource" ref="dataSource"/>
</bean>
```

通过如上方式包装数据源后，可以在项目中使用物理事务编码的方式来获得逻辑事务的支持，即支持直接从 DataSource 获取连接和释放连接，且这些连接自动支持 Spring 逻辑事务；

- **高级别解决方案：**
  - **模板类：**使用 Spring 提供的模板类，如 JdbcTemplate、HibernateTemplate 和 JpaTemplate 模板类等，而这些模板类内部其实是使用了低级别解决方案中的工具类来管理连接或会话；

Spring 提供两种编程式事务支持：直接使用 PlatformTransactionManager 实现和使用 TransactionTemplate 模板类，用于支持逻辑事务管理。

如果采用编程式事务推荐使用 TransactionTemplate 模板类和高级别解决方案。

9.3.3 使用 PlatformTransactionManager

首先让我们看下如何使用 PlatformTransactionManager 实现来进行事务管理：

1、数据源定义，此处使用第 7 章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

2、事务管理器定义（chapter9/applicationContext-jdbc.xml）：

```
<bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

3、准备测试环境：

3.1、首先准备测试时使用的 SQL：

```
package cn.javass.spring.chapter9;
//省略import
public class TransactionTest {
    //id自增主键从0开始
    private static final String CREATE_TABLE_SQL = "create table test" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
    private static final String DROP_TABLE_SQL = "drop table test";
    private static final String INSERT_SQL = "insert into test(name) values(?)";
    private static final String COUNT_SQL = "select count(*) from test";
    ……
}
```

3.2、初始化 Spring 容器

```
package cn.javass.spring.chapter9;
//省略import
public class TransactionTest {
    private static ApplicationContext ctx;
    private static PlatformTransactionManager txManager;
    private static DataSource dataSource;
    private static JdbcTemplate jdbcTemplate;
    ……
    @BeforeClass
    public static void setUpClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter9/applicationContext-jdbc.xml"};
        ctx = new ClassPathXmlApplicationContext(configLocations);
        txManager = ctx.getBean(PlatformTransactionManager.class);
        dataSource = ctx.getBean(DataSource.class);
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
    ……
}
```

3.3、使用高级别方案 JdbcTemplate 来进行事务管理器测试：

```
@Test
public void testPlatformTransactionManager() {
   DefaultTransactionDefinition def = new DefaultTransactionDefinition();
   def.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
   def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
   TransactionStatus status = txManager.getTransaction(def);
   jdbcTemplate.execute(CREATE_TABLE_SQL);
   try {
      jdbcTemplate.update(INSERT_SQL, "test");
      txManager.commit(status);
   } catch (RuntimeException e) {
      txManager.rollback(status);
   }
   jdbcTemplate.execute(DROP_TABLE_SQL);
}
```

- **DefaultTransactionDefinition**：事务定义，定义如隔离级别、传播行为等，即在本示例中隔离级别为 ISOLATION_READ_COMMITTED（提交读），传播行为为 PROPAGATION_REQUIRED（必须有事务支持，即如果当前没有事务，就新建一个事务，如果已经存在一个事务中，就加入到这个事务中）。
- **TransactionStatus**：事务状态类，通过 PlatformTransactionManager 的 getTransaction 方法根据事务定义获取；获取事务状态后，Spring 根据传播行为来决定如何开启事务；
- JdbcTemplate：通过 JdbcTemplate 对象执行相应的 SQL 操作，且自动享受到事务支持，注意事务是线程绑定的，因此事务管理器可以运行在多线程环境；
- txManager.commit(status)：提交 status 对象绑定的事务；
- txManager.rollback(status)：当遇到异常时回滚 status 对象绑定的事务。

3.4、使用低级别解决方案来进行事务管理器测试：

```
@Test
public void testPlatformTransactionManagerForLowLevel1() {
   DefaultTransactionDefinition def = new DefaultTransactionDefinition();
   def.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
   def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
   TransactionStatus status = txManager.getTransaction(def);
   Connection conn = DataSourceUtils.getConnection(dataSource);
   try {
         conn.prepareStatement(CREATE_TABLE_SQL).execute();
         PreparedStatement pstmt = conn.prepareStatement(INSERT_SQL);
         pstmt.setString(1, "test");
         pstmt.execute();
         conn.prepareStatement(DROP_TABLE_SQL).execute();
         txManager.commit(status);
   } catch (Exception e) {
         status.setRollbackOnly();
         txManager.rollback(status);
   } finally {
         DataSourceUtils.releaseConnection(conn, dataSource);
   }
}
```

低级别方案中使用 DataSourceUtils 获取和释放连接，使用 txManager 开管理事务，而且面向 JDBC 编程，比起模板类方式来繁琐和复杂的多，因此不推荐使用该方式。在此就不介绍数据源代理类使用了，需要请参考 platformTransactionManagerForLowLevelTest2 测试方法。

到此事务管理是不是还很繁琐？必须手工提交或回滚事务，有没有更好的解决方案呢？Spring 提供了 TransactionTemplate 模板类来简化事务管理。

9.3.4 使用 TransactionTemplate

TransactionTemplate 模板类用于简化事务管理，事务管理由模板类定义，而具体操作需要通过 TransactionCallback 回调接口或 TransactionCallbackWithoutResult 回调接口指定，通过调用模板类的参数类型为 TransactionCallback 或 TransactionCallbackWithoutResult 的 execute 方法来自动享受事务管理。

TransactionTemplate 模板类使用的回调接口：

- **TransactionCallback**：通过实现该接口的“T doInTransaction(TransactionStatus status) ”方法来定义需要事务管理的操作代码；
- **TransactionCallbackWithoutResult**：继承 TransactionCallback 接口，提供“void doInTransactionWithoutResult(TransactionStatus status)”便利接口用于方便那些不需要返回值的事务操作代码。

1、接下来演示一下 TransactionTemplate 模板类如何使用：

```
@Test
public void testTransactionTemplate() {//位于TransactionTest类中
   jdbcTemplate.execute(CREATE_TABLE_SQL);
   TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
   transactionTemplate.
         setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
   transactionTemplate.execute(new TransactionCallbackWithoutResult() {
         @Override
         protected void doInTransactionWithoutResult(TransactionStatus status) {
             jdbcTemplate.update(INSERT_SQL, "test");
   }});
   jdbcTemplate.execute(DROP_TABLE_SQL);
}
```

- **TransactionTemplate** ：通过new TransactionTemplate(txManager)创建事务模板类，其中构造器参数为PlatformTransactionManager实现，并通过其相应方法设置事务定义，如事务隔离级别、传播行为等，此处未指定传播行为，其默认为PROPAGATION_REQUIRED；
- TransactionCallbackWithoutResult：此处使用不带返回的回调实现，其 doInTransactionWithoutResult 方法实现中定义了需要事务管理的操作；
- transactionTemplate.execute()：通过该方法执行需要事务管理的回调。

这样是不是简单多了，没有事务管理代码，而是由模板类来完成事务管理。

注：对于抛出 Exception 类型的异常且需要回滚时，需要捕获异常并通过调用 status 对象的 setRollbackOnly()方法告知事务管理器当前事务需要回滚，如下所示：

```
try {
    //业务操作
} catch (Exception e) { //可使用具体业务异常代替
    status.setRollbackOnly();
}
```

2、前边已经演示了 JDBC 事务管理，接下来演示一下 JTA 分布式事务管理：

```
@Test
public void testJtaTransactionTemplate() {
   String[] configLocations = new String[] {
         "classpath:chapter9/applicationContext-jta-derby.xml"};
   ctx = new ClassPathXmlApplicationContext(configLocations);
   final PlatformTransactionManager jtaTXManager =
         ctx.getBean(PlatformTransactionManager.class);
   final DataSource derbyDataSource1 = ctx.getBean("dataSource1", DataSource.class);
   final DataSource derbyDataSource2 = ctx.getBean("dataSource2", DataSource.class);
   final JdbcTemplate jdbcTemplate1 = new JdbcTemplate(derbyDataSource1);
   final JdbcTemplate jdbcTemplate2 = new JdbcTemplate(derbyDataSource2);
   TransactionTemplate transactionTemplate =
         new TransactionTemplate(jtaTXManager);
   transactionTemplate.
         setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
   jdbcTemplate1.update(CREATE_TABLE_SQL);
   int originalCount = jdbcTemplate1.queryForInt(COUNT_SQL);
   try {
      transactionTemplate.execute(new TransactionCallbackWithoutResult() {
         @Override
         protected void doInTransactionWithoutResult(TransactionStatus status) {
            jdbcTemplate1.update(INSERT_SQL, "test");
            //因为数据库2没有创建数据库表因此会回滚事务
            jdbcTemplate2.update(INSERT_SQL, "test");
      }});
   } catch (RuntimeException e) {
      int count = jdbcTemplate1.queryForInt(COUNT_SQL);
      Assert.assertEquals(originalCount, count);
   }
   jdbcTemplate1.update(DROP_TABLE_SQL);
}
```

- **配置文件**：使用此前定义的 chapter9/applicationContext-jta-derby.xml；
- **jtaTXManager**： JTA 事务管理器；
- **derbyDataSource1 和 derbyDataSource2**：derby 数据源 1 和 derby 数据源 2；
- **jdbcTemplate1 和 jdbcTemplate2**：分别使用 derbyDataSource1 和 derbyDataSource2 构造的 JDBC 模板类；
- **transactionTemplate**：使用 jtaTXManager 事务管理器的事务管理模板类，其隔离级别为提交读，传播行为默认为 PROPAGATION_REQUIRED（必须有事务支持，即如果当前没有事务，就新建一个事务，如果已经存在一个事务中，就加入到这个事务中）；
- **jdbcTemplate1.update(CREATE_TABLE_SQL)**：此处只有 derbyDataSource1 所代表的数据库创建了“test”表，而 derbyDataSource2 所代表的数据库没有此表；
- **TransactionCallbackWithoutResult**：在此接口实现中定义了需要事务支持的操作：
  - **jdbcTemplate1.update(INSERT_SQL, "test")**：表示向数据库 1 中的 test 表中插入数据；
  - **jdbcTemplate2.update(INSERT_SQL, "test")**：表示向数据库 2 中的 test 表中插入数据，但数据库 2 没有此表将抛出异常，且 JTA 分布式事务将回滚；
- **Assert.assertEquals(originalCount, count)**：用来验证事务是否回滚，验证结果返回为 true，说明分布式事务回滚了。

到此我们已经会使用 PlatformTransactionManager 和 TransactionTemplate 进行简单事务处理了，那如何应用到实际项目中去呢？接下来让我们看下如何在实际项目中应用 Spring 管理事务。

接下来看一下如何将 Spring 管理事务应用到实际项目中，为简化演示，此处只定义最简单的模型对象和不完整的 Dao 层接口和 Service 层接口：

1、 首先定义项目中的模型对象，本示例使用用户模型和用户地址模型：
模型对象一般放在项目中的 model 包里。

```
package cn.javass.spring.chapter9.model;
public class UserModel {
    private int id;
    private String name;
    private AddressModel address;
    //省略 getter 和 setter
}
```

```
package cn.javass.spring.chapter9.model;
public class AddressModel {
    private int id;
    private String province;
    private String city;
    private String street;
    private int userId;
    //省略 getter 和 setter
}
```

2.1、定义 Dao 层接口：

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.UserModel;
public interface IUserService {
    public void save(UserModel user);
    public int countAll();
}
```

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.AddressModel;
public interface IAddressService {
    public void save(AddressModel address);
    public int countAll();
}
```

2.2、定义 Dao 层实现：

```
package cn.javass.spring.chapter9.dao.jdbc;
//省略 import，注意 model 要引用 chapter 包里的
public class UserJdbcDaoImpl extends NamedParameterJdbcDaoSupport
    implements IUserDao {
    private final String INSERT_SQL = "insert into user(name) values(:name)";
    private final String COUNT_ALL_SQL = "select count(*) from user";
    @Override
    public void save(UserModel user) {
        KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
        SqlParameterSource paramSource =
            new BeanPropertySqlParameterSource(user);
```

```
            getNamedParameterJdbcTemplate().update(
                    INSERT_SQL, paramSource, generatedKeyHolder);
             user.setId(generatedKeyHolder.getKey().intValue());
        }
        @Override
        public int countAll() {
            return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
        }
}
```

```
package cn.javass.spring.chapter9.dao.jdbc;
//省略 import，注意 model 要引用 chapter 包里的
public class AddressJdbcDaoImpl extends NamedParameterJdbcDaoSupport
        implements IAddressDao {
        private final String INSERT_SQL = "insert into address
                                          (province, city, street, user_id)" +
                                          "values(:province, :city, :street, :userId)";
        private final String COUNT_ALL_SQL = "select count(*) from address";
        @Override
        public void save(AddressModel address) {
            KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
            SqlParameterSource paramSource =
                    new BeanPropertySqlParameterSource(address);
            getNamedParameterJdbcTemplate().update(
                    INSERT_SQL, paramSource, generatedKeyHolder);
            address.setId(generatedKeyHolder.getKey().intValue());
        }
        @Override
        public int countAll() {
            return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
        }
```

3.1、定义 Service 层接口，一般使用“I×××Service”命名：

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.UserModel;
public interface IUserService {
        public void save(UserModel user);
        public int countAll();
}
```

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.AddressModel;
public interface IAddressService {
    public void save(AddressModel address);
    public int countAll();
}
```

3.2、定义 Service 层实现，一般使用“×××ServiceImpl”或“×××Service”命名：

```
package cn.javass.spring.chapter9.service.impl;
//省略 import，注意 model 要引用 chapter 包里的
public class AddressServiceImpl implements IAddressService {
    private IAddressDao addressDao;
    private PlatformTransactionManager txManager;
    public void setAddressDao(IAddressDao addressDao) {
        this.addressDao = addressDao;
    }
    public void setTxManager(PlatformTransactionManager txManager) {
        this.txManager = txManager;
    }
    @Override
    public void save(final AddressModel address) {
        TransactionTemplate transactionTemplate =
            TransactionTemplateUtils.getDefaultTransactionTemplate(txManager);
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                addressDao.save(address);
            }
        });
    }
    @Override
    public int countAll() {
        return addressDao.countAll();
    }
}
```

```
package cn.javass.spring.chapter9.service.impl;
//省略 import，注意 model 要引用 chapter 包里的
public class UserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    private PlatformTransactionManager txManager;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setTxManager(PlatformTransactionManager txManager) {
        this.txManager = txManager;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Override
    public void save(final UserModel user) {
        TransactionTemplate transactionTemplate =
            TransactionTemplateUtils.getDefaultTransactionTemplate(txManager);
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                userDao.save(user);
                user.getAddress().setUserId(user.getId());
                addressService.save(user.getAddress());
            }
        });
    }
    @Override
    public int countAll() {
        return userDao.countAll();
    }
}
```

Service 实现中需要 Spring 事务管理的部分应该使用 TransactionTemplate 模板类来包装执行。

4、定义 TransactionTemplateUtils，用于简化获取 TransactionTemplate 模板类，工具类一般放在 util 包中：

```
package cn.javass.spring.chapter9.util;
//省略 import
public class TransactionTemplateUtils {
    public static TransactionTemplate getTransactionTemplate(
            PlatformTransactionManager txManager,
            int propagationBehavior,
            int isolationLevel) {

        TransactionTemplate transactionTemplate =
            new TransactionTemplate(txManager);
        transactionTemplate.setPropagationBehavior(propagationBehavior);
        transactionTemplate.setIsolationLevel(isolationLevel);
        return transactionTemplate;
    }

    public static TransactionTemplate getDefaultTransactionTemplate(
            PlatformTransactionManager txManager) {
        return getTransactionTemplate(
                txManager,
                TransactionDefinition.PROPAGATION_REQUIRED,
                TransactionDefinition.ISOLATION_READ_COMMITTED);
    }
}
```

getDefaultTransactionTemplate 用于获取传播行为为 PROPAGATION_REQUIRED，隔离级别为 ISOLATION_READ_COMMITTED 的模板类。

5、数据源配置定义，此处使用第 7 章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

6、Dao 层配置定义（chapter9/dao/applicationContext-jdbc.xml）：

```
<bean id="txManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="abstractDao" abstract="true">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

```
<bean id="userDao"
        class="cn.javass.spring.chapter9.dao.jdbc.UserJdbcDaoImpl"
        parent="abstractDao"/>
<bean id="addressDao"
        class="cn.javass.spring.chapter9.dao.jdbc.AddressJdbcDaoImpl"
        parent="abstractDao"/>
```

7、Service 层配置定义（chapter9/service/applicationContext-service.xml）：

```
<bean id="userService"
       class="cn.javass.spring.chapter9.service.impl.UserServiceImpl">
      <property name="userDao" ref="userDao"/>
      <property name="txManager" ref="txManager"/>
      <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService"
        class="cn.javass.spring.chapter9.service.impl.AddressServiceImpl">
      <property name="addressDao" ref="addressDao"/>
      <property name="txManager" ref="txManager"/>
</bean>
```

8、准备测试需要的表创建语句，在 TransactionTest 测试类中添加如下静态变量：

```
private static final String CREATE_USER_TABLE_SQL =
      "create table user" +
      "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
      "name varchar(100))";
private static final String DROP_USER_TABLE_SQL = "drop table user";

private static final String CREATE_ADDRESS_TABLE_SQL =
      "create table address" +
      "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
      "province varchar(100), city varchar(100), street varchar(100), user_id int)";
private static final String DROP_ADDRESS_TABLE_SQL = "drop table address";
```

9、 测试一下吧：

```
@Test
    public void testServiceTransaction() {
        String[] configLocations = new String[] {
        "classpath:chapter7/applicationContext-resources.xml",
        "classpath:chapter9/dao/applicationContext-jdbc.xml",
        "classpath:chapter9/service/applicationContext-service.xml"};
        ApplicationContext ctx2 =
            new ClassPathXmlApplicationContext(configLocations);

        DataSource dataSource2 = ctx2.getBean(DataSource.class);
        JdbcTemplate jdbcTemplate2 = new JdbcTemplate(dataSource2);
        jdbcTemplate2.update(CREATE_USER_TABLE_SQL);
        jdbcTemplate2.update(CREATE_ADDRESS_TABLE_SQL);

        IUserService userService = ctx2.getBean("userService", IUserService.class);
        IAddressService addressService =
            ctx2.getBean("addressService", IAddressService.class);
        UserModel user = createDefaultUserModel();
        userService.save(user);
        Assert.assertEquals(1, userService.countAll());
        Assert.assertEquals(1, addressService.countAll());
        jdbcTemplate2.update(DROP_USER_TABLE_SQL);
        jdbcTemplate2.update(DROP_ADDRESS_TABLE_SQL);
}
private UserModel createDefaultUserModel() {
    UserModel user = new UserModel();
    user.setName("test");
    AddressModel address = new AddressModel();
    address.setProvince("beijing");
    address.setCity("beijing");
    address.setStreet("haidian");
    user.setAddress(address);
    return user;
}
```

从 Spring 容器中获取 Service 层对象，调用 Service 层对象持久化对象，大家有没有注意到 Spring 事务全部在 Service 层定义，为什么会在 Service 层定义，而不是 Dao

层定义呢？这是因为在服务层可能牵扯到业务逻辑，而每个业务逻辑可能调用多个 Dao 层方法，为保证这些操作的原子性，必须在 Service 层定义事务。

还有大家有没有注意到如果 Service 层的事务管理相当令人头疼，而且是侵入式的，有没有办法消除这些冗长的事务管理代码呢？这就需要 Spring 声明式事务支持，下一节将介绍无侵入式的声明式事务。

可能大家对事务定义中的各种属性有点困惑，如传播行为到底干什么用的？接下来将详细讲解一下事务属性。

9.3.5 事务属性

事务属性通过 TransactionDefinition 接口实现定义，主要有事务隔离级别、事务传播行为、事务超时时间、事务是否只读。

Spring 提供 TransactionDefinition 接口默认实现 DefaultTransactionDefinition，可以通过该实现类指定这些事务属性。

- **事务隔离级别：**用来解决并发事务时出现的问题，其使用 TransactionDefinition 中的静态变量来指定：
  - ISOLATION_DEFAULT：默认隔离级别，即使用底层数据库默认的隔离级别；
  - ISOLATION_READ_UNCOMMITTED：未提交读；
  - ISOLATION_READ_COMMITTED：提交读，一般情况下我们使用这个；
  - ISOLATION_REPEATABLE_READ：可重复读；
  - ISOLATION_SERIALIZABLE：序列化。

  可以使用 DefaultTransactionDefinition 类的 setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED)来指定隔离级别，其中此处表示隔离级别为提交读，也可以使用或 setIsolationLevelName(“ISOLATION_READ_COMMITTED”)方式指定，其中参数就是隔离级别静态变量的名字，但不推荐这种方式。
- **事务传播行为：**Spring 管理的事务是逻辑事务，而且物理事务和逻辑事务最大差别就在于事务传播行为，事务传播行为用于指定在多个事务方法间调用时，事务是如何在这些方法间传播的，Spring 共支持 7 种传播行为：
  - **Required**：必须有逻辑事务，否则新建一个事务，使用 PROPAGATION_REQUIRED 指定，表示如果当前存在一个逻辑事务，则加入该逻辑事务，否则将新建一个逻辑事务，如图 9-2 和 9-3 所示；

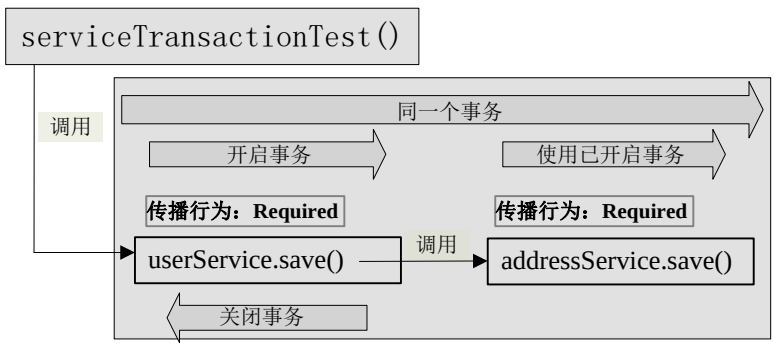

图 9-2 Required 传播行为

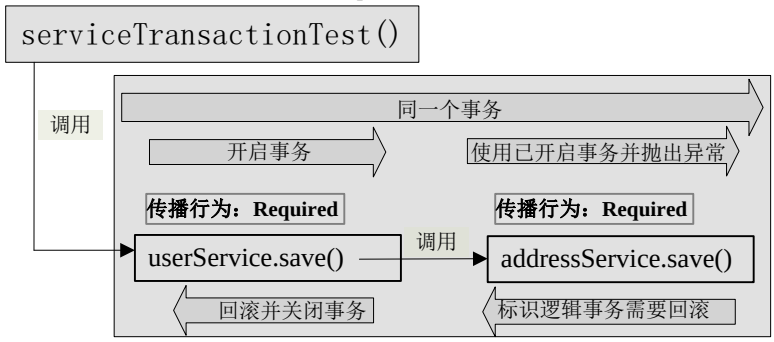

图 9-3 Required 传播行为抛出异常情况

在前边示例中就是使用的 Required 传播行为：

一、在调用 userService 对象的 save 方法时，此方法用的是 Required 传播行为且此时 Spring 事务管理器发现还没开启逻辑事务，因此 Spring 管理器觉得开启逻辑事务，

二、在此逻辑事务中调用了 addressService 对象的 save 方法，而在 save 方法中发现同样用的是 Required 传播行为,因此使用该已经存在的逻辑事务；

三、在返回到 addressService 对象的 save 方法，当事务模板类执行完毕，此时提交并关闭事务。

因此 userService 对象的 save 方法和 addressService 的 save 方法属于同一个物理事务，如果发生回滚，则两者都回滚。

接下来测试一下该传播行为如何执行吧：

一、正确提交测试，如上一节的测试，在此不再演示；

二、回滚测试，修改 AddressServiceImpl 的 save 方法片段：

```
addressDao.save(address);
```

为

```
addressDao.save(address);
//抛出异常，将标识当前事务需要回滚
throw new RuntimeException();
```

二、修改 UserServiceImpl 的 save 方法片段：

```
addressService.save(user.getAddress());
```

为

```
try {
    addressService.save(user.getAddress());//将在同一个事务内执行
} catch (RuntimeException e) {
}
```

如果该业务方法执行时事务被标记为回滚，则不管在此是否捕获该异常都将发生回滚，因为处于同一逻辑事务。

三、修改测试方法片段:

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

为如下形式：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(0, addressService.countAll());
```

Assert 断言中 countAll 方法都返回 0，说明事务回滚了，即说明两个业务方法属于同一个物理事务，即使在 userService 对象的 save 方法中将异常捕获，由于 addressService 对象的 save 方法抛出异常，即事务管理器将自动标识当前事务为需要回滚。

- **RequiresNew**：创建新的逻辑事务，使用 PROPAGATION_REQUIRES_NEW 指定，表示每次都创建新的逻辑事务（物理事务也是不同的）如图 9-4 和 9-5 所示：

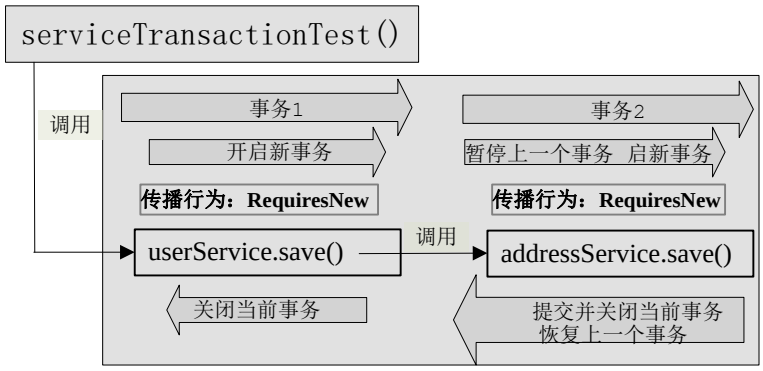

图 9-4 RequiresNew 传播行为

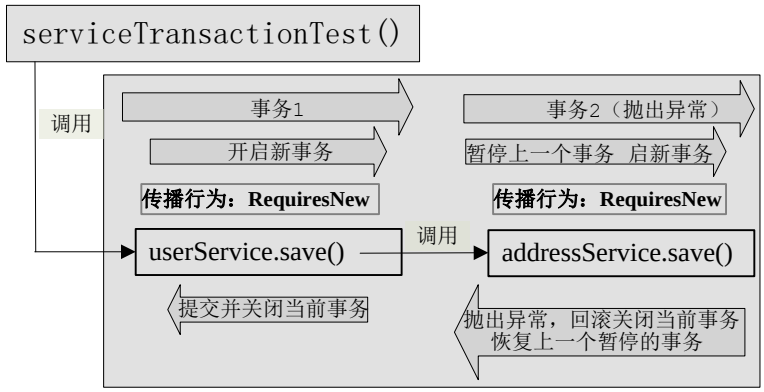

图 9-5 RequiresNew 传播行为并抛出异常

接下来测试一个该传播行为如何执行吧：

1、将如下获取事务模板方式

```
TransactionTemplate transactionTemplate =
    TransactionTemplateUtils.getDefaultTransactionTemplate(txManager);
```

替换为如下形式，表示传播行为为 RequiresNew：

```
TransactionTemplate transactionTemplate =
    TransactionTemplateUtils.getTransactionTemplate(
        txManager,
        TransactionDefinition.PROPAGATION_REQUIRES_NEW,
        TransactionDefinition.ISOLATION_READ_COMMITTED);
```

2、执行如下测试，发现执行结果是正确的：

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

3、修改 UserServiceImpl 的 save 方法片段

```
userDao.save(user);
user.getAddress().setUserId(user.getId());
addressService.save(user.getAddress());
```

为如下形式，表示 userServiceImpl 类的 save 方法将发生回滚，而 AddressServiceImpl 类的方法由于在抛出异常前执行，将成功提交事务到数据库：

```
userDao.save(user);
user.getAddress().setUserId(user.getId());
addressService.save(user.getAddress());
throw new RuntimeException();
```

4、修改测试方法片段：

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

为如下形式：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

Assert 断言中调用 userService 对象 countAll 方法返回 0，说明该逻辑事务作用域回滚，而调用 addressService 对象的 countAll 方法返回 1，说明该逻辑事务作用域正确提交。因此这是不正确的行为，因为用户和地址应该是一一对应的，不应该发生这种情况，因此此处正确的传播行为应该是 Required。

该传播行为执行流程（正确提交情况）：

一、当执行 userService 对象的 save 方法时，由于传播行为是 RequiresNew，因此创建一个新的逻辑事务（物理事务也是不同的）；

二、当执行到 addressService 对象的 save 方法时，由于传播行为是 RequiresNew，因此首先暂停上一个逻辑事务并创建一个新的逻辑事务（物理事务也是不同的）；

三、addressService 对象的 save 方法执行完毕后，提交逻辑事务（并提交物理事务）并重新恢复上一个逻辑事务，继续执行 userService 对象的 save 方法内的操作；

四、最后 userService 对象的 save 方法执行完毕，提交逻辑事务（并提交物理事务）；

五、userService 对象的 save 方法和 addressService 对象的 save 方法不属于同一个逻辑事务且也不属于同一个物理事务。

- **Supports**：支持当前事务，使用 PROPAGATION_SUPPORTS 指定，指如果当前存在逻辑事务，就加入到该逻辑事务，如果当前没有逻辑事务，就以非事务方式执行，如图 9-6 和 9-7 所示：

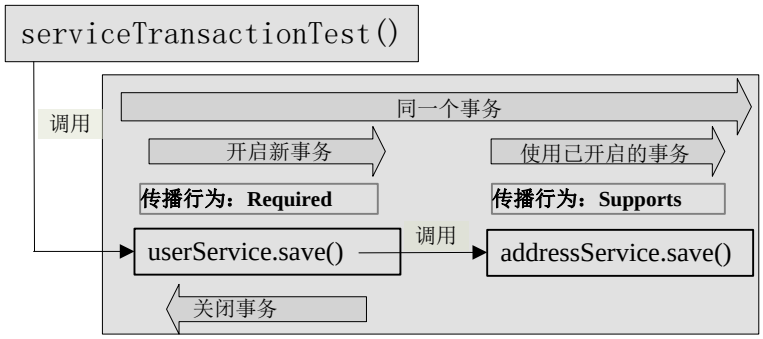

图 9-6 Required+Supports 传播行为

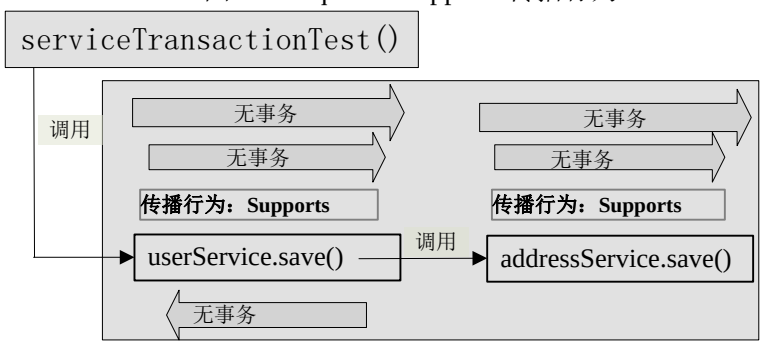

图 9-7 Supports+Supports 传播行为

- **NotSupported**：不支持事务，如果当前存在事务则暂停该事务，使用 PROPAGATION_NOT_SUPPORTED 指定，即以非事务方式执行，如果当前存在逻辑事务，就把当前事务暂停，以非事务方式执行，如图 9-8 和 9-9 所示：

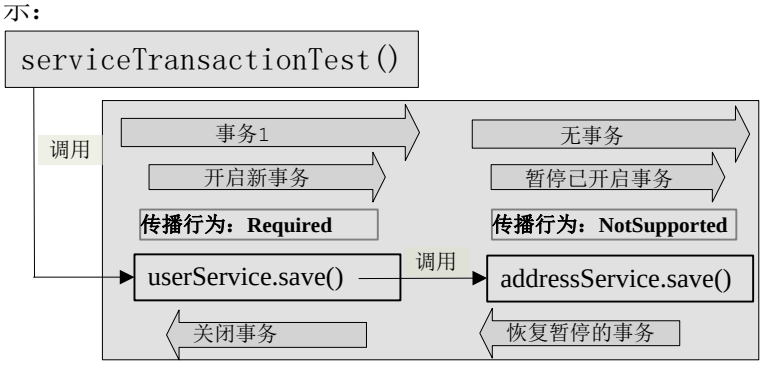

图 9-8 Required+NotSupported 传播行为

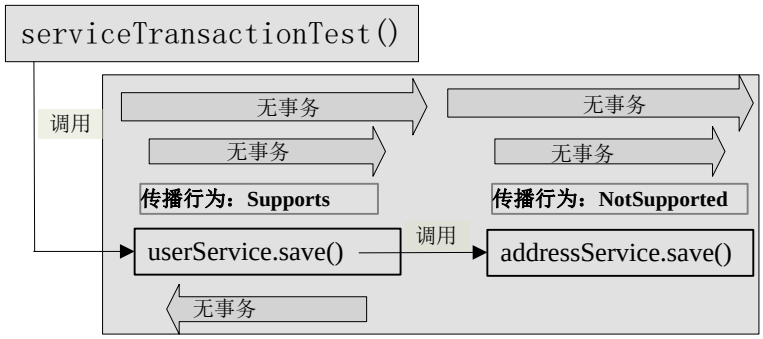

图 9-9 Supports+NotSupported 传播行为

- **Mandatory**：必须有事务，否则抛出异常，使用 PROPAGATION_MANDATORY 指定，使用当前事务执行，如果当前没有事务，则抛出异常（IllegalTransactionStateException），如图 9-10 和 9-11 所示：

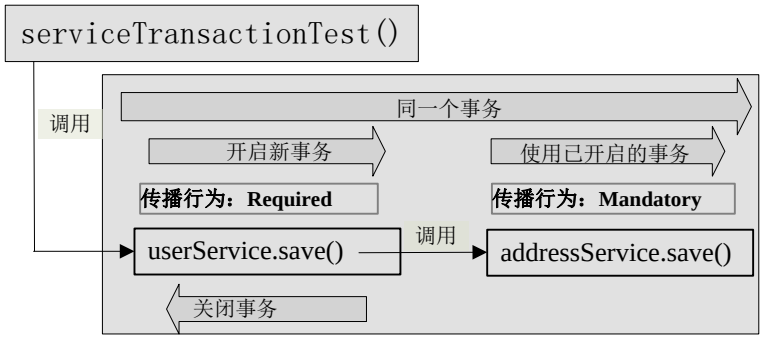

图 9-10 Required+Mandatory 传播行为

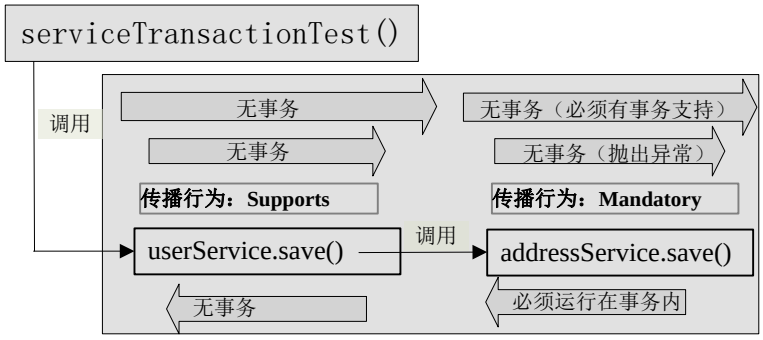

图 9-11 Supports+Mandatory 传播行为

- **Never**：不支持事务，如果当前存在是事务则抛出异常，使用 PROPAGATION_NEVER 指定，即以非事务方式执行，如果当前存在事务，则抛出异常（IllegalTransactionStateException），如图 9-12 和 9-13 所示：

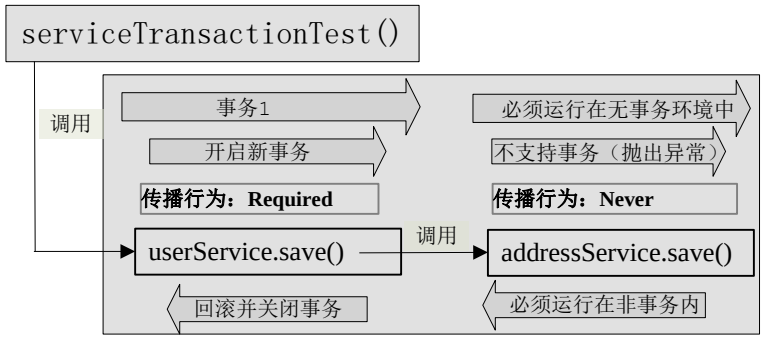

图 9-12 Required+Never 传播行为

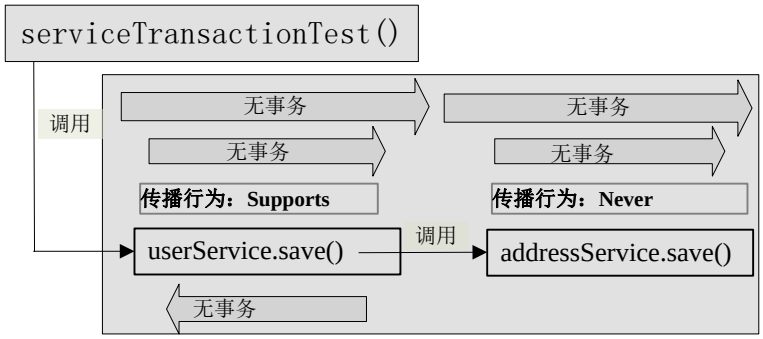

图 9-13 Supports+Never 传播行为

- **Nested**：嵌套事务支持，使用 PROPAGATION_NESTED 指定，如果当前存在事务，则在嵌套事务内执行，如果当前不存在事务，则创建一个新的事务，嵌套事务使用数据库中的保存点来实现，即嵌套事务回滚不影响外部事务，但外部事务回滚将导致嵌套事务回滚，如图 9-14 和 9-15 所示：

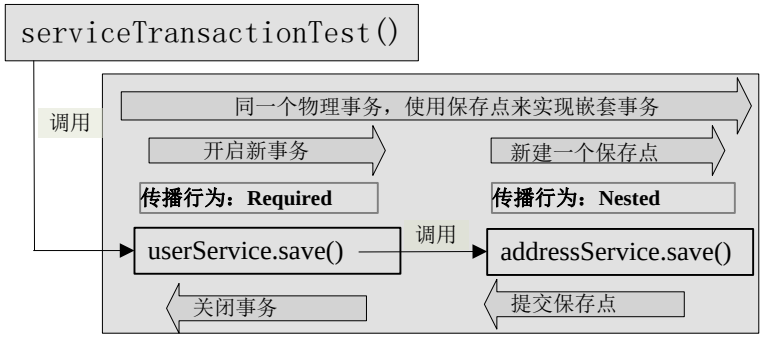

图 9-14 Required+Nested 传播行为

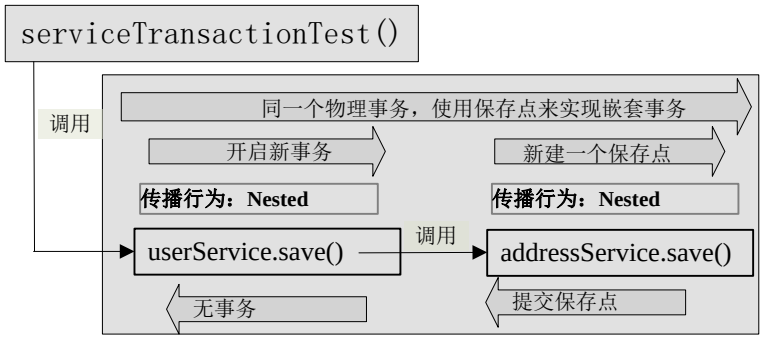

图 9-15 Nested+Nested 传播行为

Nested 和 RequiresNew 的区别：

1、 RequiresNew 每次都创建新的独立的物理事务，而 Nested 只有一个物理事务；

2、 Nested 嵌套事务回滚或提交不会导致外部事务回滚或提交，但外部事务回滚将导致嵌套事务回滚，而 RequiresNew 由于都是全新的事务，所以之间是无关联的；

3、 Nested 使用 JDBC 3 的保存点实现，即如果使用低版本驱动将导致不支持嵌套事务。

使用嵌套事务，必须确保具体事务管理器实现的 nestedTransactionAllowed 属性为 true，否则不支持嵌套事务，如 DataSourceTransactionManager 默认支持，而 HibernateTransactionManager 默认不支持，需要我们来开启。

对于事务传播行为我们只演示了 Required 和 RequiresNew，其他传播行为类似，如果对这些事务传播行为不太会使用，请参考 chapter9 包下的 TransactionTest 测试类中的 testPropagation 方法，方法内有详细示例。

- **事务超时：**设置事务的超时时间，单位为秒，默认为-1 表示使用底层事务的超时时间；
  - 使用如 setTimeout(100)来设置超时时间，如果事务超时将抛出 org.springframework.transaction.TransactionTimedOutException 异常并将当前事务标记为应该回滚，即超时后事务被自动回滚；
  - 可以使用具体事务管理器实现的 defaultTimeout 属性设置默认的事务超时时间，如 DataSourceTransactionManager. setDefaultTimeout(10)。

- **事务只读：**将事务标识为只读，只读事务不修改任何数据；
  - 对于 JDBC 只是简单的将连接设置为只读模式，对于更新将抛出异常；
  - 而对于一些其他 ORM 框架有一些优化作用，如在 Hibernate 中，Spring 事务管理器将执行“session.setFlushMode(FlushMode.MANUAL)”即指定 Hibernate 会话在只读事务模式下不用尝试检测和同步持久对象的状态的更新。
  - 如果使用设置具体事务管理的 validateExistingTransaction 属性为 true（默认 false），将确保整个事务传播链都是只读或都不是只读，如图 9-16 是正确的

事务只读设置，而图 9-17 是错误的事务只读设置：

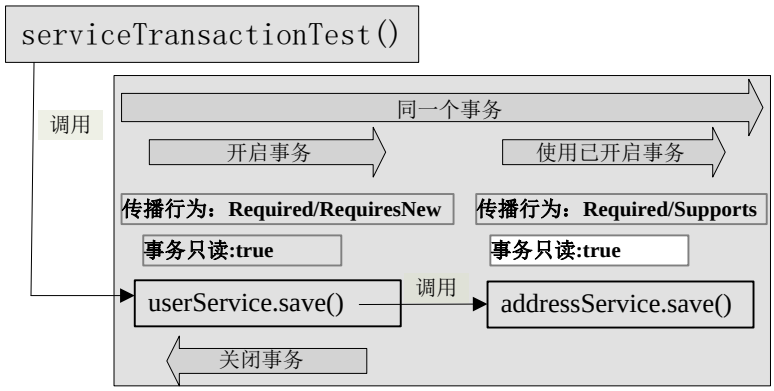

图 9-16 正确的事务只读设置

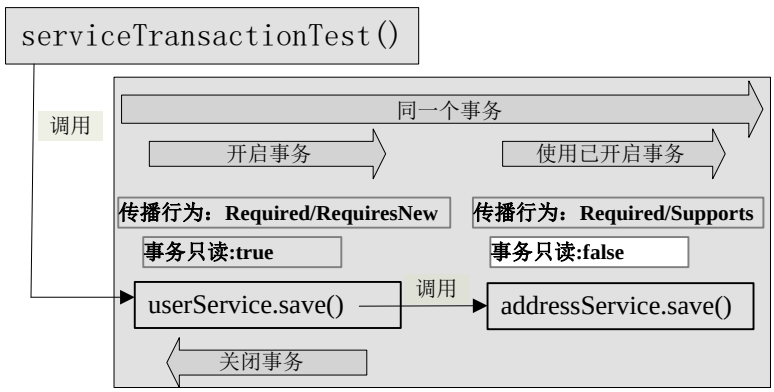

图 9-17 错误的事务只读设置

如图 10-17，对于错误的事务只读设置将抛出 IllegalTransactionStateException 异常，并伴随“Participating transaction with definition [……] is not marked as read-only……”信息，表示参与的事务只读属性设置错误。

大家有没有感觉到编程式实现事务管理是不是很繁琐冗长，重复，而且是侵入式的，因此发展到这 Spring 决定使用配置方式实现事务管理。

9.3.6 配置方式实现事务管理

在 Spring2.x 之前为了解决编程式事务管理的各种不好问题，Spring 提出使用配置方式实现事务管理，配置方式利用代理机制实现，即使有 TransactionProxyFactoryBean 类来为目标类代理事务管理。

接下来演示一下具体使用吧：

1、重新定义业务类实现，在业务类中无需显示的事务管理代码：

```
package cn.javass.spring.chapter9.service.impl;
//省略 import
public class ConfigAddressServiceImpl implements IAddressService {
    private IAddressDao addressDao;
    public void setAddressDao(IAddressDao addressDao) {
        this.addressDao = addressDao;
    }
    @Override
    public void save(final AddressModel address) {
        addressDao.save(address);
    }
    //countAll 方法实现不变
}
```

```
package cn.javass.spring.chapter9.service.impl;
//省略 import
public class ConfigUserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Override
    public void save(final UserModel user) {
        userDao.save(user);
        user.getAddress().setUserId(user.getId());
        addressService.save(user.getAddress());
    }
    //countAll 方法实现不变
}
```

从以上业务类中可以看出，没有事务管理的代码，即没有侵入式的代码。

2、在 chapter9/service/applicationContext-service.xml 配置文件中添加如下配置：

2.1、首先添加目标类定义：

```
<bean id="targetUserService"
        class="cn.javass.spring.chapter9.service.impl.ConfigUserServiceImpl">
      <property name="userDao" ref="userDao"/>
      <property name="addressService" ref="targetAddressService"/>
</bean>
<bean id="targetAddressService"
        class="cn.javass.spring.chapter9.service.impl.ConfigAddressServiceImpl">
      <property name="addressDao" ref="addressDao"/>
</bean>
```

2.2、配置 TransactionProxyFactoryBean 类：

```
<bean id="transactionProxyParent"
        class="org.springframework.transaction.interceptor.
                TransactionProxyFactoryBean"
        abstract="true">
      <property name="transactionManager" ref="txManager"/>
      <property name="transactionAttributes">
          <props>
              <prop key="save*">
                  PROPAGATION_REQUIRED,
                  ISOLATION_READ_COMMITTED,
                  timeout_10,
                  -Exception,
                  +NoRollBackException
              </prop>
              <prop key="*">
                  PROPAGATION_REQUIRED,
                  ISOLATION_READ_COMMITTED,
                  readOnly
              </prop>
          </props>
      </property>
</bean>
```

- **TransactionProxyFactoryBean**：用于为目标业务类创建代理的 Bean；
- **abstract="true"**：表示该 Bean 是抽象的，用于去除重复配置；
- **transactionManager**：事务管理器定义；
- **transactionAttributes**：表示事务属性定义：

- **PROPAGATION_REQUIRED,ISOLATION_READ_COMMITTED,timeout_10,-Exception,+NoRollBackException**：事务属性定义，Required 传播行为，提交读隔离级别，事务超时时间为 10 秒，将对所有 Exception 异常回滚，而对于抛出 NoRollBackException 异常将不发生回滚而是提交；
- **PROPAGATION_REQUIRED,ISOLATION_READ_COMMITTED,readOnly**：事务属性定义，Required 传播行为，提交读隔离级别，事务是只读的，且只对默认的 RuntimeException 异常回滚；
- **<prop key="save*">**：表示将代理以 save 开头的方法，即当执行到该方法时会为该方法根据事务属性配置来开启/关闭事务；
- **<prop key="*">**：表示将代理其他所有方法，但需要注意代理方式，默认是 JDK 代理，只有 public 方法能代理；

注：事务属性的传播行为和隔离级别使用 TransactionDefinition 静态变量名指定；事务超时使用“timeout_超时时间”指定，事务只读使用“readOnly”指定，需要回滚的异常使用“-异常”指定，不需要回滚的异常使用“+异常”指定，默认只对 RuntimeException 异常回滚。

需要特别注意“-异常”和“+异常”中“异常”只是真实异常的部分名，内部使用如下方式判断：

```
//真实抛出的异常.name.indexOf(配置中指定的需要回滚/不回滚的异常名)
exceptionClass.getName().indexOf(this.exceptionName)
```

因此异常定义时需要特别注意，配置中定义的异常只是真实异常的部分名。

2.3、定义代理 Bean：

```
<bean id="proxyUserService" parent="transactionProxyParent">
    <property name="target" ref="targetUserService"/>
</bean>
<bean id="proxyAddressService" parent="transactionProxyParent">
    <property name="target" ref="targetAddressService"/>
</bean>
```

代理 Bean 通过集成抽象 Bean“transactionProxyParent”，并通过 target 属性设置目标 Bean，在实际使用中应该使用该代理 Bean。

3、修改测试方法并测试该配置方式是否好用：

将 TransactionTest 类的 testServiceTransaction 测试方法拷贝一份命名为 testConfigTransaction：

并在 testConfigTransaction 测试方法内将：

```
IUserService userService =
        ctx2.getBean("userService", IUserService.class);
IAddressService addressService =
        ctx2.getBean("addressService", IAddressService.class);
```

替换为：

```
IUserService userService =
        ctx2.getBean("proxyUserService ", IUserService.class);
IAddressService addressService =
        ctx2.getBean("proxyAddressService ", IAddressService.class);
```

4、执行测试，测试正常通过，说明该方式能正常工作，当调用 save 方法时将匹配到“**<prop key="save*">**”定义，而 countAll 将匹配到“**<prop key="save*">**”定义，底层代理会应用相应定义中的事务属性来创建或关闭事务。

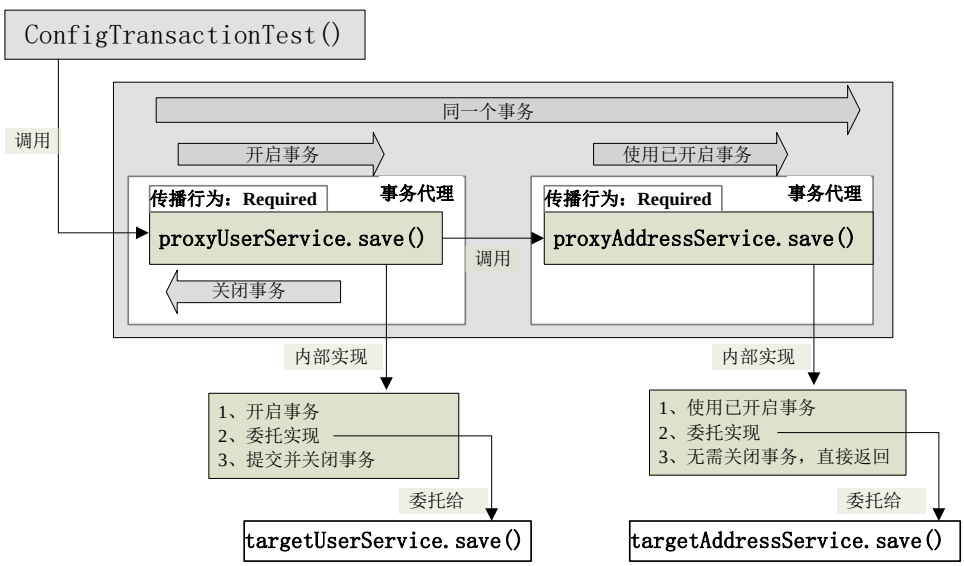

图 9-18 代理方式实现事务管理

如图 9-18，代理方式实现事务管理只是将硬编码的事务管理代码转移到代理中去由代理实现，在代理中实现事务管理。

注：**在代理模式下，默认只有通过代理对象调用的方法才能应用相应的事务属性，而在目标方法内的“自我调用”是不会应用相应的事务属性的，即被调用方法不会应用相应的事务属性，而是使用调用方法的事务属性。**

如图9-19所示，在目标对象targetUserService的save方法内调用事务方法“this.otherTransactionMethod()”将不会应用配置的传播行为RequriesNew，开启新事务，

而是使用save方法的已开启事务，如果非要这样使用如下方式实现：

1、 修改 TransactionProxyFactoryBean 配置定义，添加 exposeProxy 属性为 true；

2、 在业务方法内通过代理对象调用相应的事务方放，如“**((IUserService)AopContext.currentProxy()).otherTransactionMethod()**”即可应用配置的事务属性。

3、 使用这种方式属于侵入式，不推荐使用，除非必要。

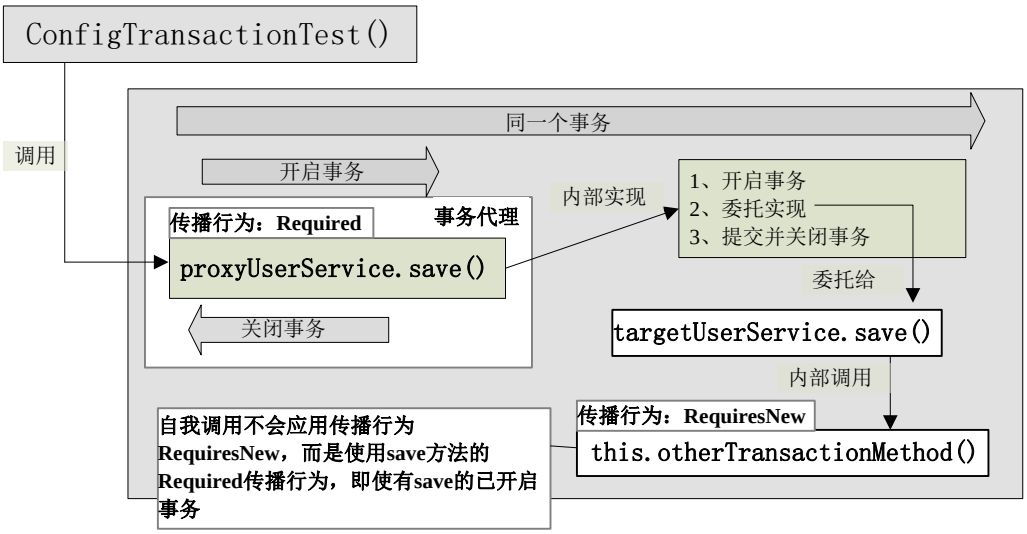

图 9-19 代理方式下的自我调用

配置方式也好麻烦啊，每个业务实现都需要配置一个事务代理，发展到这，Spring 想出更好的解决方案，Spring2.0 及之后版本提出使用新的“<tx:tags/>”方式配置事务，从而无需为每个业务实现配置一个代理。

9.4 声明式事务

9.4.1 声明式事务概述

从上节编程式实现事务管理可以深刻体会到编程式事务的痛苦，即使通过代理配置方式也是不小的工作量。

本节将介绍声明式事务支持，使用该方式后最大的获益是简单，事务管理不再是令人痛苦的，而且此方式属于无侵入式，对业务逻辑实现无影响。

接下来先来看看声明式事务如何实现吧。

9.4.2 声明式实现事务管理

1、定义业务逻辑实现，此处使用 ConfigUserServiceImpl 和 ConfigAddressServiceImpl：

2、定义配置文件（chapter9/service/ applicationContext-service-declare.xml）：

2.1、XML 命名空间定义，定义用于事务支持的 tx 命名空间和 AOP 支持的 aop 命名空间：

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/tx
            http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

2.2、业务实现配置，非常简单，使用以前定义的非侵入式业务实现：

```
<bean id="userService"
      class="cn.javass.spring.chapter9.service.impl.ConfigUserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService"
      class="cn.javass.spring.chapter9.service.impl.ConfigAddressServiceImpl">
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.3、事务相关配置：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*"
                   propagation="REQUIRED"
                   isolation="READ_COMMITTED"/>
        <tx:method name="*"
                   propagation="REQUIRED"
                   isolation="READ_COMMITTED"
                   read-only="true"/>
    </tx:attributes>
</tx:advice>
```

```
<aop:config>
    <aop:pointcut id="serviceMethod"
                  expression="execution(* cn..chapter9.service..*.*(..))"/>
    <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice"/>
</aop:config>
```

- <tx:advice>：事务通知定义，用于指定事务属性，其中“transaction-manager”属性指定事务管理器，并通过< tx:attributes >指定具体需要拦截的方法；
  - <tx:method name="save*">：表示将拦截以 save 开头的方法，被拦截的方法将应用配置的事务属性：propagation="REQUIRED"表示传播行为是 Required，isolation="READ_COMMITTED"表示隔离级别是提交读；
  - <tx:method name="*">：表示将拦截其他所有方法，被拦截的方法将应用配置的事务属性：propagation="REQUIRED"表示传播行为是 Required，isolation="READ_COMMITTED" 表示隔离级别是提交读，read-only="true"表示事务只读；
- <aop:config>：AOP 相关配置：
  - <aop:pointcut/>：切入点定义，定义名为"serviceMethod"的 aspectj 切入点，切入点表达式为"execution(* cn..chapter9.service..*.*(..))"表示拦截 cn 包及子包下的 chapter9. service 包及子包下的任何类的任何方法；
  - <aop:advisor>：Advisor 定义，其中切入点为 serviceMethod，通知为 txAdvice。

从配置中可以看出，将对 cn 包及子包下的 chapter9. service 包及子包下的任何类的任何方法应用“txAdvice”通知指定的事务属性。

3、修改测试方法并测试该配置方式是否好用：

将 TransactionTest 类的 testServiceTransaction 测试方法拷贝一份命名为 testDeclareTransaction：

并在 testDeclareTransaction 测试方法内将：

```
classpath:chapter9/service/applicationContext-service.xml"
```

替换为：

```
classpath:chapter9/service/applicationContext-service-declare.xml"
```

4、执行测试，测试正常通过，说明该方式能正常工作，当调用 save 方法时将匹配到事务通知中定义的“<tx:method name="save*">”中指定的事务属性，而调用 countAll 方法时将匹配到事务通知中定义的“<tx:method name="*">”中指定的事务属性。

声明式事务是如何实现事务管理的呢？还记不记得 TransactionProxyFactoryBean 实现配置式事务管理，配置式事务管理是通过代理方式实现，而声明式事务管理同样是通过 AOP

代理方式实现。

声明式事务通过 AOP 代理方式实现事务管理，利用环绕通知 TransactionInterceptor 实现事务的开启及关闭，而 TransactionProxyFactoryBean 内部也是通过该环绕通知实现的，因此可以认为是<tx:tags/>帮你定义了 TransactionProxyFactoryBean，从而简化事务管理。

了解了实现方式后，接下来详细学习一下配置吧：

9.4.4 <tx:advice/>配置详解

声明式事务管理通过配置<tx:advice/>来定义事务属性，配置方式如下所示：

```
<tx:advice id="……" transaction-manager="……">
    <tx:attributes>
        <tx:method name="……"
                   propagation=" REQUIRED"
                   isolation="READ_COMMITTED"
                   timeout="-1"
                   read-only="false"
                   no-rollback-for=""
                   rollback-for=""/>
        ……
    </tx:attributes>
</tx:advice>
```

- **<tx:advice>：**id 用于指定此通知的名字， transaction-manager 用于指定事务管理器，默认的事务管理器名字为“transactionManager”；
- <tx:method>：用于定义事务属性即相关联的方法名；
  - **name：**定义与事务属性相关联的方法名，将对匹配的方法应用定义的事务属性，可以使用“*”通配符来匹配一组或所有方法，如“save*”将匹配以 save 开头的方法，而“*”将匹配所有方法；
    - **pro**pagation：事务传播行为定义，默认为“REQUIRED”，表示 Required，其值可以通过 TransactionDefinition 的静态传播行为变量的“PROPAGATION_”后边部分指定，如“TransactionDefinition.PROPAGATION_REQUIRED”可以使用“REQUIRED”指定；
  - isolation：事务隔离级别定义；默认为“DEFAULT”，其值可以通过 TransactionDefinition 的静态隔离级别变量的“ISOLATION_”后边部分指定，如“TransactionDefinition. ISOLATION_DEFAULT”可以使用“DEFAULT”指定：
  - **timeout：**事务超时时间设置，单位为秒，默认-1，表示事务超时将依赖于底层事务系统；

- **read-only**：事务只读设置，默认为 false，表示不是只读；
- **rollback-for**：需要触发回滚的异常定义，以“，”分割，默认任何 RuntimeException 将导致事务回滚，而任何 Checked Exception 将不导致事务回滚；异常名字定义和 TransactionProxyFactoryBean 中含义一样
- **no-rollback-for**：不被触发进行回滚的 Exception(s)；以“，”分割；异常名字定义和 TransactionProxyFactoryBean 中含义一样；

记不记得在配置方式中为了解决“自我调用”而导致的不能设置正确的事务属性问题，使用“**((IUserService)AopContext.currentProxy()).otherTransactionMethod()**”方式解决，在声明式事务要得到支持需要使用<aop:config expose-proxy="true">来开启。

9.4.5 多事务语义配置及最佳实践

什么是多事务语义？说白了就是为不同的 Bean 配置不同的事务属性，因为我们项目中不可能就几个 Bean，而可能很多，这可能需要为 Bean 分组，为不同组的 Bean 配置不同的事务语义。在 Spring 中，可以通过配置多切入点和多事务通知并通过不同方式组合使用即可。

1、首先看下声明式事务配置的最佳实践吧：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED" />
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="create*" propagation="REQUIRED" />
        <tx:method name="insert*" propagation="REQUIRED" />
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="merge*" propagation="REQUIRED" />
        <tx:method name="del*" propagation="REQUIRED" />
        <tx:method name="remove*" propagation="REQUIRED" />
        <tx:method name="put*" propagation="REQUIRED" />
        <tx:method name="get*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="count*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
```

```
            <tx:method name="list*" propagation="SUPPORTS" read-only="true" />
            <tx:method name="*" propagation="SUPPORTS" read-only="true" />
        </tx:attributes>
</tx:advice>
<aop:config>
        <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service.*.*(..))" />
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>
```

该声明式事务配置可以应付常见的 CRUD 接口定义，并实现事务管理，我们只需修改切入点表达式来拦截我们的业务实现从而对其应用事务属性就可以了，如果还有更复杂的事务属性直接添加即可，即即如果我们有一个 batchSaveOrUpdate 方法需要“REQUIRES_NEW”事务传播行为，则直接添加如下配置即可：

```
<tx:method name="batchSaveOrUpdate" propagation="REQUIRES_NEW" />
```

2、接下来看一下多事务语义配置吧，声明式事务最佳实践中已经配置了通用事务属性，因此可以针对需要其他事务属性的业务方法进行特例化配置：

```
<tx:advice id="noTxAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="*" propagation="NEVER" />
        </tx:attributes>
</tx:advice>
<aop:config>
        <aop:pointcut id="noTxPointcut" expression="execution(* cn.javass..util.*.*())" />
        <aop:advisor advice-ref="noTxPointcut" pointcut-ref="noTxAdvice" />
</aop:config>
```

该声明将对切入点匹配的方法所在事务应用“Never”传播行为。

多事务语义配置时，切入点一定不要叠加，否则将应用两次事务属性，造成不必要的错误及麻烦。

9.4.6 @Transactional 实现事务管理

对声明式事务管理，Spring 提供基于@Transactional 注解方式来实现，但需要 Java 5+。

注解方式是最简单的事务配置方式，可以直接在 Java 源代码中声明事务属性，且对于每一个业务类或方法如果需要事务都必须使用此注解。

接下来学习一下注解事务的使用吧：

1、定义业务逻辑实现：

```
package cn.javass.spring.chapter9.service.impl;
//省略 import
public class AnnotationUserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Transactional(propagation=Propagation.REQUIRED,
                   isolation=Isolation.READ_COMMITTED)
    @Override
    public void save(final UserModel user) {
        userDao.save(user);
        user.getAddress().setUserId(user.getId());
        addressService.save(user.getAddress());
    }
    @Transactional(propagation=Propagation.REQUIRED,
                   isolation=Isolation.READ_COMMITTED, readOnly=true)
    @Override
    public int countAll() {
        return userDao.countAll();
    }
}
```

```
package cn.javass.spring.chapter9.service.impl;
//省略 import
@Transactional(propagation=Propagation.REQUIRED,
               isolation=Isolation.READ_COMMITTED)
public class AnnotationAddressServiceImpl implements IAddressService {
    private IAddressDao addressDao;
    public void setAddressDao(IAddressDao addressDao) {
        this.addressDao = addressDao;
    }
```

```
    @Override
    public void save(final AddressModel address) {
        addressDao.save(address);
        throw new RuntimeException();//将导致事务回滚
    }
    @Transactional(propagation=Propagation.REQUIRED,
                   isolation=Isolation.READ_COMMITTED, readOnly=true)
    @Override
    public int countAll() {
        return addressDao.countAll();
    }
}
```

2、定义配置文件（chapter9/service/ applicationContext-service-annotation.xml）：

2.1、XML 命名空间定义，定义用于事务支持的 tx 命名空间和 AOP 支持的 aop 命名空间：

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

2.2、业务实现配置，非常简单，使用以前定义的非侵入式业务实现：

```
<bean id="userService"
      class="cn.javass.spring.chapter9.service.impl.ConfigUserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService"
      class="cn.javass.spring.chapter9.service.impl.ConfigAddressServiceImpl">
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.3、事务相关配置：

```
<tx:annotation-driven transaction-manager="txManager"/>
```

使用如上配置已支持声明式事务。

3、修改测试方法并测试该配置方式是否好用：

将 TransactionTest 类的 testServiceTransaction 测试方法拷贝一份命名为 testAnntationTransactionTest：

将测试代码片段：

```
classpath:chapter9/service/applicationContext-service.xml"
```

替换为：

```
classpath:chapter9/service/applicationContext-service-annotation.xml"
```

将测试代码段

```
userService.save(user);
```

替换为：

```
try {
        userService.save(user);
        Assert.fail();
} catch (RuntimeException e) {
}
        Assert.assertEquals(0, userService.countAll());
        Assert.assertEquals(0, addressService.countAll());
```

4、执行测试，测试正常通过，说明该方式能正常工作，因为在 AnnotationAddressServiceImpl 类的 save 方法中抛出异常，因此事务需要回滚，所以两个 countAll 操作都返回 0。

9.4.7 @Transactional 配置详解

Spring 提供的<tx:annotation-driven/>用于开启对注解事务管理的支持，从而能识别 Bean 类上的@Transactional 注解元数据，其具有以下属性：

- transaction-manager：指定事务管理器名字，默认为 transactionManager，当使用其他名字时需要明确指定；
- proxy-target-class：表示将使用的代码机制，默认 false 表示使用 JDK 代理，如果为 true 将使用 CGLIB 代理
- order：定义事务通知顺序，默认 Ordered.LOWEST_PRECEDENCE，表示将顺序

决定权交给 AOP 来处理。

Spring 使用@Transaction 来指定事务属性，可以在接口、类或方法上指定，如果类和方法上都指定了@Transaction，则方法上的事务属性被优先使用，具体属性如下：

- value：指定事务管理器名字，默认使用<tx:annotation-driven/>指定的事务管理器，用于支持多事务管理器环境；
- **propagation**：指定事务传播行为，默认为 Required，使用 Propagation.REQUIRED 指定；
- **isolation**：指定事务隔离级别，默认为“DEFAULT”，使用 Isolation.DEFAULT 指定；
- **readOnly**：指定事务是否只读，默认 false 表示事务非只读；
- **timeout**：指定事务超时时间，以秒为单位，默认-1 表示事务超时将依赖于底层事务系统；
- **rollbackFor**：指定一组异常类，遇到该类异常将回滚事务；
- **rollbackForClassname**：指定一组异常类名字，其含义与<tx:method>中的 rollback-for 属性语义完全一样；
- **noRollbackFor**：指定一组异常类，即使遇到该类异常也将提交事务，即不回滚事务；
- **noRollbackForClassname**：指定一组异常类名字，其含义与<tx:method>中的 no-rollback-for 属性语义完全一样；

Spring 提供的@Transaction 注解事务管理内部同样利用环绕通知 TransactionInterceptor 实现事务的开启及关闭。

使用@Transaction 注解事务管理需要特别注意以下几点：

- 如果在接口、实现类或方法上都指定了@Transactional 注解，则优先级顺序为方法>实现类>接口；
- 建议只在实现类或实现类的方法上使用@Transactional，而不要在接口上使用，这是因为如果使用 JDK 代理机制是没问题，因为其使用基于接口的代理；而使用使用 CGLIB 代理机制时就会遇到问题，因为其使用基于类的代理而不是接口，这是**因为接口上的@Transactional 注解是“不能继承的”**；
- 在 JDK 代理机制下，“自我调用”同样不会应用相应的事务属性，其语义和<tx:tags>中一样；
- 默认只对 RuntimeException 异常回滚；
- 在使用 Spring 代理时，默认只有在 public 可见度的方法的@Transactional 注解才是有效的，其它可见度（protected、private、包可见）的方法上即使有@Transactional 注解也不会应用这些事务属性的，Spring 也不会报错，如果你非要使用非公共方法注解事务管理的话，可考虑使用 AspectJ。

9.4.9 与其他 AOP 通知协作

Spring 声明式事务实现其实就是 Spring AOP+线程绑定实现，利用 AOP 实现开启和关闭事务，利用线程绑定（ThreadLocal）实现跨越多个方法实现事务传播。

由于我们不可能只使用一个事务通知，可能还有其他类型事务通知，而且如果这些通知中需要事务支持怎么办？这就牵扯到通知执行顺序的问题上了，因此如果可能与其他 AOP 通知协作的话，而且这些通知中需要使用声明式事务管理支持，事务通知应该具有最高优先级。

9.4.10 声明式 or 编程式

编程式事务时不推荐的，即使有很少事务操作，Spring 发展到现在，没有理由使用编程式事务，只有在为了深入理解 Spring 事务管理才需要学习编程式事务使用。

推荐使用声明式事务，而且强烈推荐使用<tx:tags>方式的声明式事务，因为其是无侵入代码的，可以配置模板化的事务属性并运用到多个项目中。

而@Transaction 注解事务，可以使用，不过作者更倾向于使用<tx:tags>声明式事务。

能保证项目正常工作的事务配置就是最好的。

9.4.11 混合事务管理

所谓混合事务管理就是混合多种数据访问技术使用，如混合使用 Spring JDBC + Hibernate，接下来让我们学习一下常见混合事务管理：

1、 Hibernate + Spring JDBC/iBATIS：使用 HibernateTransactionManager 即可支持；

2、 JPA + Spring JDBC/iBATIS：使用 JpaTransactionManager 即可支持；

3、 JDO + Spring JDBC/iBATIS：使用 JtaTransactionManager 即可支持；

混合事务管理最大问题在于如果我们使用第三方 ORM 框架，如 Hibernate，会遇到一级及二级缓存问题，尤其是二级缓存可能造成如使用 Spring JDBC 和 Hibernate 查询出来的数据不一致等。

因此不建议使用这种混合使用和混合事务管理。