阅读笔记5

第六章　事务管理

事务是双方的行为
全有或者全无 的操作被称为事务
事务允许你将几个操作合成一个要么全部发生 要全部不发生的工作单元
事务确保了数据或资源免于处在不一致的状态中 如果没有它们的话 可能会损坏数据或者导致应用程序业务规则的不一致性

事务的特性　原子性 一致性 隔离性 持久性

- **原子性**（Atomic）：事务是由一个或多个活动所组成的一个工作单元。原子性确保事务中的所有操作全部发生或全部不发生。如果所有的活动都成功了，事务也就成功了。如果任意一个活动失败了，整个事务也失败并回滚。
- **一致性**（Consistent）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态。现实的数据不应该被损坏。
- **隔离性**（Isolated）：事务允许多个用户对相同的数据进行操作，每个用户的操作不会与其他用户纠缠在一起。因此，事务应该被彼此隔离，避免发生同步读写相同数据的事情（注意的是，隔离性往往涉及到锁定数据库中的行或表）。
- **持久性**（Durable）：一旦事务完成，事务的结果应该持久化，这样就能从任何的系统崩溃中恢复过来。这一般会涉及将结果存储到数据库或其他形式的持久化存储中。

当上面任一步骤失败时 所有的步骤的操作结果都会被取消 从而保证事务的原子性
原子性保证系统数据永远不会处于不一致或者部分完成的状态来保证了一致性
隔离性同样保证了一致性
最后做出的结果是持久化的
　不用担心事务的结果会被丢失

Spring对事务管理的支持
Spring对编码式和声明式事务管理的支持
Spring与EJB感觉很像 但是完全不一样
Spring是通过回调机制将事务的实现从事务性的代码中抽象出来
如果应用程序只使用一种资源的话 Spring可以使用持久化机制本身的事务性支持
包括了JDBC hibernate　JPA
如果横跨多个资源的话 Spring就会使用第三方的JTA来实现分布式事务

编码式事务允许用户在代码中精确定义事务的边界 而声明式事务有助于用户将操作与事务规则进行解耦
选择编码式还是声明式很多程度上是在细粒度和易用性之间仅仅想权衡
当通过编码实现事务管理时 能精确控制事务的边界 开始于结束完成取决于需求

关于事务管理器 Spring提供了多个事务管理器 它们将事务委托给JTA或其他持久化机制所提供的平台相关的事务实现

表 6.1　　Spring 为每种场景都提供了事务管理器

事务管理器（org.framework.*）	使用场景
jca.cci.connection.CciLocalTransactionManager	使用 Spring 对 Java EE 连接器架构（Java EE Connector Architecture,JCA）和通用客户端接口（Common Client Interface,CCI）提供支持
jdbc.datasource.DataSourceTransactionManager	用于 Spring 对 JDBC 抽象的支持，也可用于使用 iBATIS 进行持久化的场景
jms.connection.JmsTransactionManager	用于 JMS 1.1+
jms.connection.JmsTransactionManager102	用于 JMS 1.0.2
orm.hibernate3.HibernateTransactionManager	用于 Hibernate 3 进行持久化
orm.jdo.JdoTransactionManager	用于 JDO 进行持久化
orm.jpa.JpaTransactionManager	用于 Java 持久化 API（Java Persistence API，JPA）进行持久化
transaction.jta.JtaTransactionManager	需要分布式事务或者没有其他的事务管理器满足需求
transaction.jta.OC4JJtaTransactionManager	用于 Oracle 的 OC4J JEE 容器
transaction.jta.WebLogicJtaTransactionManager	需要使用分布式事务并且应用程序运行于 WebLogic 中
transaction.jta.WebSphereUowTransactionManager	需要 WebSphere 中 UOWManager 所管理的事务

JDBC事务
下面贴代码

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

Hibernate 事务

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

</bean>
```

下面是JPA的事务

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

除了将事务用于JPA的操作中 这个JpaTransactionManager还支持简单的JDBC操作之中
这个JDBC使用的database与entityManagerFactory使用的database必须是相同的
首先 为了做到这一点 JpaTransactionManager必须要装配一个JpaDialect的实现

```
<bean id="JpaDialect" class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect"></bean>
```

然后我们必须将jpaDialect这个Bean装载到JpaTransactionManager

这样这个JpaDialect的实现就可以让这个JpaTransactionManager匹配JPA与JDBC俩种方式进行访问工作了

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="emf" />
    <property name="jpaDialect" ref="JpaDialect"/>
</bean>
```

我们还可以装载特定厂商的JpaDialect实现如：EclipseLinkJpaDialect HibernateJpaDialect,OpenJPaDialect 和ToplinkJpaDialect 提供对JPA/JDBC的支持

而DefaultJpaDialect是不行的
这种方式也是可以的 这个我先把代码放出来 我下面的使用的具体事务配置的时候 使用的是JDBC的配置数据源 但是我换了这个之后也是一样成功 没有阻碍的 这本书说的关键在于必须安装一个JpaDialect 但是我发现不用的话也可以 可能是版本的原因 现在的JPA已经全面支持JDBC了

```
<bean id="spitterService"
class="Spitter.SpitterServiceImpl">
 <property name="transactionTemplate">
        <bean class="org.springframework.transaction.support.TransactionTemplate">
               <property name="transactionManager" ref="transactionManager"></property>
        </bean>
 </property>
 <property name="spitterDao" ref="spitterDao"></property>
 </bean>


<!-- <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTrans
actionManager">
        <property name="dataSource" ref="dataSource"/>
 </bean>-->
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="emf" />
        <property name="jpaDialect" ref="JpaDialect"></property>
 </bean>
 <bean id="JpaDialect" class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect"></bea
n>

 <bean id="emf" class=
         "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
 </bean>


 <bean id="jpaVendorAdapter"
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="database" value="MYSQL" />
        <property name="showSql" value="true"/>
        <property name="generateDdl" value="false"/>
        <property name="databasePlatform"
value="org.hibernate.dialect.MySQLDialect" />
 </bean>



 <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/spitter"></property>
        <property name="username" value="root"/>
        <property name="password" value="liuziye"/>
 </bean>
 <bean id="spitterDao" class="Spitter.SpitterDaoImpl">
        <property name="dataSource" ref="dataSource"/>
 </bean>
```

JTA（分布式）
我们还可以使用JTA来实现来匹配事务源列如：俩个不同的数据库
你就可以使用JTA

```
<bean id="transactionManager2" class="org.springframework.transaction.jta.JtaTransactionManag
er">
     <property name="transactionManager" value="java://TransactionManager"/>
</bean>
```

JTA有一套特殊的API在事务与数据源之间 这个transactionManager提供了关于JPA事务的管理来进行寻找相关的JNDI

关于事务的级别选择 更细度的是选择编码式
而声明式只能在方法级别声明事务的边界 细度较低

```
public void saveSpittle(Spittle spittle) {
  spittle.setWhen(new Date());
spitterDao.saveSpittle(spittle);
}
```

以这个方法为例
添加事务的一种方式saveSpittle（）方法中直接通过编码使用Spring的TransactionTemplate来添加事务性边界 就像Spring的其他模板类

这是代码

```
@Service("spitterService")
/*@Transactional(propagation=Propagation.REQUIRED)*/
public class SpitterServiceImpl implements SpitterService {

public void saveSpittle(  final SpittleEntity spittle) {
transactionTemplate.execute(new TransactionCallback<Void>() {
public Void doInTransaction(TransactionStatus txStatus) {
try {
spitterDao.saveSpittleEntity(spittle);
} catch (RuntimeException e) {
          txStatus.setRollbackOnly();
          throw e;
}
return null;
}
    });
}
  TransactionTemplate transactionTemplate;
SpitterDao spitterDao;

  public void setTransactionTemplate(TransactionTemplate transactionTemplate) {
this.transactionTemplate = transactionTemplate;
}


public void setSpitterDao(SpitterDao spitterDao) {
this.spitterDao = spitterDao;
}

public static void main(String[] args) {
    SpittleEntity spittle=new SpittleEntity();
spittle.setId((long) 3);
spittle.setSpitter("3");
spittle.setText("ceshi");
spittle.setWhen(new Date(1015,10,21));
ApplicationContext context=new ClassPathXmlApplicationContext("Spring-tx.xml");
SpitterServiceImpl spitterService= (SpitterServiceImpl) context.getBean("spitterService");
spitterService.saveSpittle(spittle);
System.out.println("1");
}
```

还有相关的Spring xml配置 在这里的transactionmanager至于为什么要加载datasource这个Bean的原因是 我们这个transactionManager
是通过调用java.sql.Connection来管理事务的 而java.sql.Connection是通过DataSource来获取到的 所以我们需要配置这个datasouce 如果不配置就会无法完成事务的回滚与提交 插一句 什么样的事务配置什么样的具体的事务管理器

```
<bean id="spitterService"
class="Spitter.SpitterServiceImpl">
<property name="transactionTemplate">
       <bean class="org.springframework.transaction.support.TransactionTemplate">
              <property name="transactionManager" ref="transactionManager"></property>
       </bean>
</property>
<property name="spitterDao" ref="spitterDao"></property>
```

```
</bean>


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


<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
	<property name="driverClassName" value="com.mysql.jdbc.Driver"/>
	<property name="url" value="jdbc:mysql://localhost:3306/spitter"></property>
	<property name="username" value="root"/>
	<property name="password" value="liuziye"/>
</bean>
<bean id="spitterDao" class="Spitter.SpitterDaoImpl">
	<property name="dataSource" ref="dataSource"/>
</bean>
```

以及其他的 比如SpitterDaoImpl的这个类的具体实现 就是照抄以前的JDBC具体实现 也粘贴一下吧

```
private static final String SQL_INSERT_SPITTER = "insert into spittle (id,spitter,text) value
s (?,?, ?)";


@Override
public void saveSpittleEntity( SpittleEntity spittle) {
    getSimpleJdbcTemplate().update(SQL_INSERT_SPITTER,
spittle.getId(),
spittle.getSpitter(),
spittle.getText())
;
}
```

但是这个可以看出这个编码式的事务控制是入侵式的
通常情况下 是不需要这么精细的事务边界控制
这个时候就有了另一种事务 声明式事务

声明式事务管理无论是哪种实现方式 都会有5个属性 分别是传播行为 是否只读 隔离级别 事务超时 回调规则
Spring定义了7种不同的传播行为
传播规则定义了何时创建一个事务或何时使用已有的事务
这里有一个慕课网的讲师表格 暂且粘贴过来

事务传播行为类型	说明
PROPAGATION_REQUIRED	支持当前事务，如果不存在 就新建一个
PROPAGATION_SUPPORTS	支持当前事务，如果不存在，就不使用事务
PROPAGATION_MANDATORY	支持当前事务，如果不存在，抛出异常
PROPAGATION_REQUIRES_NEW	如果有事务存在，挂起当前事务，创建一个新的事务
PROPAGATION_NOT_SUPPORTED	以非事务方式运行，如果有事务存在，挂起当前事务
PROPAGATION_NEVER	以非事务方式运行，如果有事务存在，抛出异常
PROPAGATION_NESTED	如果当前事务存在，则嵌套事务执行

这个1-3是一组
就是如果第一个service有事务的话，就使用事务，如果没有事务的话，就按不同的来
然后 4-6
如果service 1 service2不在一个事务里面的话，如果service1有的话，1就新建一个，2就挂起，3就报异常
然后最后一个
是一种嵌套的事务 可以自己控制 就是service1执行完之后，设置保存点，然后如果service2发生异常之后，我们可以让service2回滚到保

存点的位置，或者最初始的状态，

大概是这么个意思

关于隔离级别
脏读
不可重复读
幻读
http://blog.csdn.net/d8111/article/details/2595635
http://uule.iteye.com/blog/1109647
有俩篇博客讲解这个
关于隔离级别一共在Spring中提供了5个
ISOLATION_DEFAULT 使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED 允许读取尚未提交的数据 会有脏读 不可重复读 幻读 但是效率最高
ISOLATION_READ_COMMITTED 允许读取并发事务已经提交的数据 可以阻止脏读 但是不可以阻止不可重复读 幻读
ISOLATION_REPEATABLE_READ 对同一字段多次读取结果是一致的 除非数据是被本身修改 可以阻止脏读 不可重复读 但是可能出现幻读
ISOLATION_SERIALIZABLE 完全服从ACID的隔离级别 完全锁定事务相关的数据库表来实现

只读的属性 **来标识这个是否为只读事务 如果只读事务的话是对后端的是数据库进行读操作**
数据库可以利用事务的只读属性来进行一些特定的优化
只读优化是在事务启动的时候由数据库来实施的 只有对那些具备启动一个新事务
（PROPAGATION_REQUIRED,PROPAGATION_REQUIRES_NEW以及PROPAGTION_NESTED）的方法来说
才有意义

事务超时 为了使应用程序很好的运行 事务不能运行太长时间 所以 事务就有了超时的限制 这个跟只读一样 由于超时时钟是在事务开始时启动的 所以
只有对那些具备启动一个新事务（PROPAGATION_REQUIRED,PROPAGATION_REQUIRES_NEW以及PROPAGTION_NESTED）的方法来说
才有意义

回滚规则
默认是遇到运行时异常才会回滚 遇到检测型的异常不会回滚 所以你可以自己定义规则来声明遇到检测型的也进行回滚 同样 也可以设置运行时不回滚

表 6.4　事务五边形的 5 个方面（见图 6.3）通过该元素的属性来指定

隔离级别	含义
isolation	指定事务的隔离级别
propagation	定义事务的传播规则
read-only	指定事务为只读
回滚规则： rollback-for no-rollback-for	rollback-for 指定事务对于哪些检查型异常应当回滚而不提交 no-rollback-for 指定事务对于哪些异常应当继续运行而不回滚
timeout	对于长时间运行的事务定义超时时间

关于在xml中的声明 注意AOP 也是要进行命名空间的声明的 因为一些声明式的事务配置元素依赖于部分Spring的AOP配置元素
同时 我们为了省去TransactionProxyFactoryBean的特殊Bean的配置 我们加入tx的命名空间
于是就是下面这个样子

```
<?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:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.or
g/schema/beans/spring-beans.xsd">
```

在其中我们定义了一个<tx:advice>

```
<tx:advice>
	<tx:attributes>
		<tx:method name="save*" propagation="REQUIRED"/>
		<tx:method name="*" propagation="SUPPORTS" read-only="true"/>
	</tx:attributes>
```

```
</tx:advice>
```

这个配置信息的意思是save开头的方法是用REQUIRED的级别 也就是不存在就创建
如果是其他的方法 的话就如果不存在就不创建事务 跟上面的表示对应的 包括这个只读
但是这样还不够 这只是个事务 我们必须将这个加入切面

```
<aop:config>
       <aop:advisor  pointcut="execution(* Spitter.*(..))"
advice-ref="txadvice"/>
</aop:config>
```

这样就好了

这个有问题。。
注意JAR包 实测半天出错的问题 还是JAR包的原因 注意JAR包 注意框架 在有的时候 jar包也是有bug的 有的版本上有bug 所以说有的时候还是需要
换一下版本来试试原因

我们还可以用注解的方式来进行事务的声明

```
@Transactional(propagation=Propagation.SUPPORTS,readOnly = false)
```

这个是声明在我们要进行事务管理的方法上的 如果我们声明在了类上 根据约定大于规定的原则 所以这个声明的这个方法特殊 但是其他的没有声明

事务注解的类就会自动用类注解的事务方式来使用 另外我们还需要在xml中进行如下定义

```
<tx:annotation-driven/>
```

这里要注意 我们如果我们的方法是从接口声明来得到的一定要用接口来接 不然会出现转型错误 因为这是内部采用了代理的缘故

如果不是采用接口的方式的话就不用这样了 但是Spring推荐的就是面向接口编程 使用接口定义方法好处多多 尽量使用吧