[Spring事务配置的五种方式和spring里面事务的传播属性和事务隔离级别](http://blog.csdn.net/it_man/article/details/5074371)

标签： [spring](http://www.csdn.net/tag/spring)[bean](http://www.csdn.net/tag/bean)[nested](http://www.csdn.net/tag/nested)[class](http://www.csdn.net/tag/class)[aop](http://www.csdn.net/tag/aop)[数据库](http://www.csdn.net/tag/%e6%95%b0%e6%8d%ae%e5%ba%93)

2009-12-25 12:25 38424人阅读 [评论](http://blog.csdn.net/it_man/article/details/5074371#comments)(11) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/it_man/article/details/5074371#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

开源技术(如Struts/spring/Hibernate等)（70） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

**Spring事务配置的五种方式**

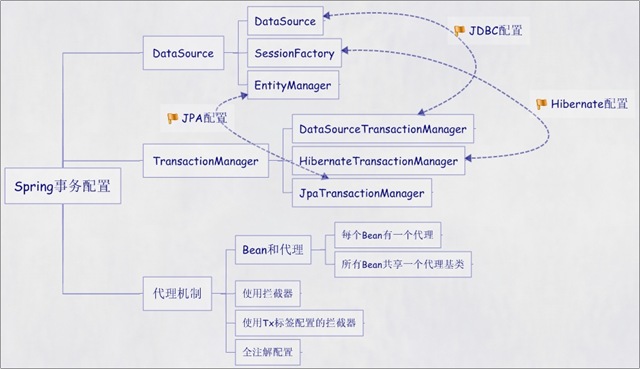
前段时间对Spring的事务配置做了比较深入的研究，在此之间对Spring的事务配置虽说也配置过，但是一直没有一个清楚的认识。通过这次的学习发觉Spring的事务配置只要把思路理清，还是比较好掌握的。

    总结如下：

    Spring配置文件中关于事务配置总是由三个组成部分，分别是DataSource、TransactionManager和代理机制这三部分，无论哪种配置方式，一般变化的只是代理机制这部分。

    DataSource、TransactionManager这两部分只是会根据数据访问方式有所变化，比如使用Hibernate进行数据访问时，DataSource实际为SessionFactory，TransactionManager的实现为HibernateTransactionManager。

    具体如下图：

[](http://www.blogjava.net/images/blogjava_net/robbie/WindowsLiveWriter/Spring_9C9C/Spring%E4%BA%8B%E5%8A%A1%E9%85%8D%E7%BD%AE%20(2).jpg)

    根据代理机制的不同，总结了五种Spring事务的配置方式，配置文件如下：

    第一种方式：每个Bean都有一个代理

<?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:context="http://www.springframework.org/schema/context"  
    xmlns:aop="http://www.springframework.org/schema/aop"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans   
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
           http://www.springframework.org/schema/context  
           http://www.springframework.org/schema/context/spring-context-2.5.xsd  
           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">  
  
    <bean id="sessionFactory"    
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">    
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />    
        <property name="configurationClass" value="org.hibernate.cfg.AnnotationConfiguration" />  
    </bean>    
  
    <!-- 定义事务管理器（声明式的事务） -->    
    <bean id="transactionManager"  
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
      
    <!-- 配置DAO -->  
    <bean id="userDaoTarget" class="com.bluesky.spring.dao.UserDaoImpl">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
      
    <bean id="userDao"    
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">    
           <!-- 配置事务管理器 -->    
           <property name="transactionManager" ref="transactionManager" />       
        <property name="target" ref="userDaoTarget" />    
         <property name="proxyInterfaces" value="com.bluesky.spring.dao.GeneratorDao" />  
        <!-- 配置事务属性 -->    
        <property name="transactionAttributes">    
            <props>    
                <prop key="\*">PROPAGATION\_REQUIRED</prop>  
            </props>    
        </property>    
    </bean>    
</beans>

    第二种方式：所有Bean共享一个代理基类

<?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:context="http://www.springframework.org/schema/context"  
    xmlns:aop="http://www.springframework.org/schema/aop"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans   
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
           http://www.springframework.org/schema/context  
           http://www.springframework.org/schema/context/spring-context-2.5.xsd  
           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">  
  
    <bean id="sessionFactory"    
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">    
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />    
        <property name="configurationClass" value="org.hibernate.cfg.AnnotationConfiguration" />  
    </bean>    
  
    <!-- 定义事务管理器（声明式的事务） -->    
    <bean id="transactionManager"  
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
      
    <bean id="transactionBase"    
            class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"    
            lazy-init="true" abstract="true">    
        <!-- 配置事务管理器 -->    
        <property name="transactionManager" ref="transactionManager" />    
        <!-- 配置事务属性 -->    
        <property name="transactionAttributes">    
            <props>    
                <prop key="\*">PROPAGATION\_REQUIRED</prop>    
            </props>    
        </property>    
    </bean>      
     
    <!-- 配置DAO -->  
    <bean id="userDaoTarget" class="com.bluesky.spring.dao.UserDaoImpl">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
      
    <bean id="userDao" parent="transactionBase" >    
        <property name="target" ref="userDaoTarget" />     
    </bean>  
</beans>

第三种方式：使用拦截器

<?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:context="http://www.springframework.org/schema/context"  
    xmlns:aop="http://www.springframework.org/schema/aop"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans   
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
           http://www.springframework.org/schema/context  
           http://www.springframework.org/schema/context/spring-context-2.5.xsd  
           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">  
  
    <bean id="sessionFactory"    
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">    
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />    
        <property name="configurationClass" value="org.hibernate.cfg.AnnotationConfiguration" />  
    </bean>    
  
    <!-- 定义事务管理器（声明式的事务） -->    
    <bean id="transactionManager"  
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>   
     
    <bean id="transactionInterceptor"    
        class="org.springframework.transaction.interceptor.TransactionInterceptor">    
        <property name="transactionManager" ref="transactionManager" />    
        <!-- 配置事务属性 -->    
        <property name="transactionAttributes">    
            <props>    
                <prop key="\*">PROPAGATION\_REQUIRED</prop>    
            </props>    
        </property>    
    </bean>  
        
    <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">    
        <property name="beanNames">    
            <list>    
                <value>\*Dao</value>  
            </list>    
        </property>    
        <property name="interceptorNames">    
            <list>    
                <value>transactionInterceptor</value>    
            </list>    
        </property>    
    </bean>    
    
    <!-- 配置DAO -->  
    <bean id="userDao" class="com.bluesky.spring.dao.UserDaoImpl">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
</beans>

第四种方式：使用tx标签配置的拦截器

<?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:context="http://www.springframework.org/schema/context"  
    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.org/schema/beans/spring-beans-2.5.xsd  
           http://www.springframework.org/schema/context  
           http://www.springframework.org/schema/context/spring-context-2.5.xsd  
           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd  
           http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">  
  
    <context:annotation-config />  
    <context:component-scan base-package="com.bluesky" />  
  
    <bean id="sessionFactory"    
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">    
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />    
        <property name="configurationClass" value="org.hibernate.cfg.AnnotationConfiguration" />  
    </bean>    
  
    <!-- 定义事务管理器（声明式的事务） -->    
    <bean id="transactionManager"  
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
  
    <tx:advice id="txAdvice" transaction-manager="transactionManager">  
        <tx:attributes>  
            <tx:method name="\*" propagation="REQUIRED" />  
        </tx:attributes>  
    </tx:advice>  
      
    <aop:config>  
        <aop:pointcut id="interceptorPointCuts"  
            expression="execution(\* com.bluesky.spring.dao.\*.\*(..))" />  
        <aop:advisor advice-ref="txAdvice"  
            pointcut-ref="interceptorPointCuts" />          
    </aop:config>        
</beans>

第五种方式：全注解

<?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:context="http://www.springframework.org/schema/context"  
    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.org/schema/beans/spring-beans-2.5.xsd  
           http://www.springframework.org/schema/context  
           http://www.springframework.org/schema/context/spring-context-2.5.xsd  
           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd  
           http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">  
  
    <context:annotation-config />  
    <context:component-scan base-package="com.bluesky" />  
  
    <tx:annotation-driven transaction-manager="transactionManager"/>  
  
    <bean id="sessionFactory"    
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">    
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />    
        <property name="configurationClass" value="org.hibernate.cfg.AnnotationConfiguration" />  
    </bean>    
  
    <!-- 定义事务管理器（声明式的事务） -->    
    <bean id="transactionManager"  
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
      
</beans>

此时在DAO上需加上@Transactional注解，如下：

package com.bluesky.spring.dao;  
  
import java.util.List;  
  
import org.hibernate.SessionFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;  
import org.springframework.stereotype.Component;  
  
import com.bluesky.spring.domain.User;  
  
@Transactional  
@Component("userDao")  
public class UserDaoImpl extends HibernateDaoSupport implements UserDao {  
  
    public List<User> listUsers() {  
        return this.getSession().createQuery("from User").list();  
    }  
      
    http://www.blogjava.net/Images/dot.gif  
}

**spring**里面事务的传播属性和**事务隔离级别**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 一、Propagation （事务的传播属性）  Propagation ：　　key属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。有以下选项可供使用：  PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。  PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。  PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。  PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。  PROPAGATION\_NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。  PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。  1： PROPAGATION\_REQUIRED  加入当前正要执行的事务不在另外一个事务里，那么就起一个新的事务  比如说，ServiceB.methodB的事务级别定义为PROPAGATION\_REQUIRED, 那么由于执行ServiceA.methodA的时候，  ServiceA.methodA已经起了事务，这时调用ServiceB.methodB，ServiceB.methodB看到自己已经运行在ServiceA.methodA  的事务内部，就不再起新的事务。而假如ServiceA.methodA运行的时候发现自己没有在事务中，他就会为自己分配一个事务。  这样，在ServiceA.methodA或者在ServiceB.methodB内的任何地方出现异常，事务都会被回滚。即使ServiceB.methodB的事务已经被  提交，但是ServiceA.methodA在接下来fail要回滚，ServiceB.methodB也要回滚  2： PROPAGATION\_SUPPORTS  如果当前在事务中，即以事务的形式运行，如果当前不再一个事务中，那么就以非事务的形式运行  3： PROPAGATION\_MANDATORY  必须在一个事务中运行。也就是说，他只能被一个父事务调用。否则，他就要抛出异常  4： PROPAGATION\_REQUIRES\_NEW  这个就比较绕口了。 比如我们设计ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_REQUIRES\_NEW，  那么当执行到ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起，ServiceB.methodB会起一个新的事务，等待ServiceB.methodB的事务完成以后，  他才继续执行。他与PROPAGATION\_REQUIRED 的事务区别在于事务的回滚程度了。因为ServiceB.methodB是新起一个事务，那么就是存在  两个不同的事务。如果ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚，ServiceB.methodB是不会回滚的。如果ServiceB.methodB失败回滚，  如果他抛出的异常被ServiceA.methodA捕获，ServiceA.methodA事务仍然可能提交。  5： PROPAGATION\_NOT\_SUPPORTED  当前不支持事务。比如ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED ，而ServiceB.methodB的事务级别是PROPAGATION\_NOT\_SUPPORTED ，  那么当执行到ServiceB.methodB时，ServiceA.methodA的事务挂起，而他以非事务的状态运行完，再继续ServiceA.methodA的事务。  6： PROPAGATION\_NEVER  不能在事务中运行。假设ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED， 而ServiceB.methodB的事务级别是PROPAGATION\_NEVER ，  那么ServiceB.methodB就要抛出异常了。  7： PROPAGATION\_NESTED  理解Nested的关键是savepoint。他与PROPAGATION\_REQUIRES\_NEW的区别是，PROPAGATION\_REQUIRES\_NEW另起一个事务，将会与他的父事务相互独立，  而Nested的事务和他的父事务是相依的，他的提交是要等和他的父事务一块提交的。也就是说，如果父事务最后回滚，他也要回滚的。  而Nested事务的好处是他有一个savepoint。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  ServiceA {  /\*\*  \* 事务属性配置为 PROPAGATION\_REQUIRED  \*/  void methodA() {  try {  //savepoint  ServiceB.methodB(); //PROPAGATION\_NESTED 级别  } catch (SomeException) {  // 执行其他业务, 如 ServiceC.methodC();  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  也就是说ServiceB.methodB失败回滚，那么ServiceA.methodA也会回滚到savepoint点上，ServiceA.methodA可以选择另外一个分支，比如  ServiceC.methodC，继续执行，来尝试完成自己的事务。  但是这个事务并没有在EJB标准中定义。  二、Isolation Level(事务隔离等级):  1、Serializable：最严格的级别，事务串行执行，资源消耗最大；  2、REPEATABLE READ：保证了一个事务不会修改已经由另一个事务读取但未提交（回滚）的数据。避免了“脏读取”和“不可重复读取”的情况，但是带来了更多的性能损失。  3、READ COMMITTED:大多数主流数据库的默认事务等级，保证了一个事务不会读到另一个并行事务已修改但未提交的数据，避免了“脏读取”。该级别适用于大多数系统。  4、Read Uncommitted：保证了读取过程中不会读取到非法数据。隔离级别在于处理多事务的并发问题。  我们知道并行可以提高数据库的吞吐量和效率，但是并不是所有的并发事务都可以并发运行，这需要查看数据库教材的可串行化条件判断了。  这里就不阐述。  我们首先说并发中可能发生的3中不讨人喜欢的事情  1： Dirty reads--读脏数据。也就是说，比如事务A的未提交（还依然缓存）的数据被事务B读走，如果事务A失败回滚，会导致事务B所读取的的数据是错误的。  2： non-repeatable reads--数据不可重复读。比如事务A中两处读取数据-total-的值。在第一读的时候，total是100，然后事务B就把total的数据改成 200，事务A再读一次，结果就发现，total竟然就变成200了，造成事务A数据混乱。  3： phantom reads--幻象读数据，这个和non-repeatable reads相似，也是同一个事务中多次读不一致的问题。但是non-repeatable reads的不一致是因为他所要取的数据集被改变了（比如total的数据），但是phantom reads所要读的数据的不一致却不是他所要读的数据集改变，而是他的条件数据集改变。比如Select account.id where account.name="ppgogo\*",第一次读去了6个符合条件的id，第二次读取的时候，由于事务b把一个帐号的名字由"dd"改成"ppgogo1"，结果取出来了7个数据。     |  |  |  |  | | --- | --- | --- | --- | |  | Dirty reads | non-repeatable reads | phantom reads | | Serializable | 不会 | 不会 | 不会 | | REPEATABLE READ | 不会 | 不会 | 会 | | READ COMMITTED | 不会 | 会 | 会 | | Read Uncommitted | 会 | 会 | 会 |     三、readOnly  事务属性中的readOnly标志表示对应的事务应该被最优化为只读事务。  这是一个最优化提示。在一些情况下，一些事务策略能够起到显著的最优化效果，例如在使用Object/Relational映射工具（如：Hibernate或TopLink）时避免dirty checking（试图“刷新”）。  四、Timeout  在事务属性中还有定义“timeout”值的选项，指定事务超时为几秒。在JTA中，这将被简单地传递到J2EE服务器的事务协调程序，并据此得到相应的解释 |

**spring里面事务的传播属性和事务隔离级别**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 一、Propagation （事务的传播属性）  Propagation ：　　key属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。有以下选项可供使用：PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。  PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。  PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。  PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。  PROPAGATION\_NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。  PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。  1： PROPAGATION\_REQUIRED  加入当前正要执行的事务不在另外一个事务里，那么就起一个新的事务  比如说，ServiceB.methodB的事务级别定义为PROPAGATION\_REQUIRED, 那么由于执行ServiceA.methodA的时候，  ServiceA.methodA已经起了事务，这时调用ServiceB.methodB，ServiceB.methodB看到自己已经运行在ServiceA.methodA  的事务内部，就不再起新的事务。而假如ServiceA.methodA运行的时候发现自己没有在事务中，他就会为自己分配一个事务。  这样，在ServiceA.methodA或者在ServiceB.methodB内的任何地方出现异常，事务都会被回滚。即使ServiceB.methodB的事务已经被  提交，但是ServiceA.methodA在接下来fail要回滚，ServiceB.methodB也要回滚  2： PROPAGATION\_SUPPORTS  如果当前在事务中，即以事务的形式运行，如果当前不再一个事务中，那么就以非事务的形式运行  3： PROPAGATION\_MANDATORY  必须在一个事务中运行。也就是说，他只能被一个父事务调用。否则，他就要抛出异常  4： PROPAGATION\_REQUIRES\_NEW  这个就比较绕口了。 比如我们设计ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_REQUIRES\_NEW，  那么当执行到ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起，ServiceB.methodB会起一个新的事务，等待ServiceB.methodB的事务完成以后，  他才继续执行。他与PROPAGATION\_REQUIRED 的事务区别在于事务的回滚程度了。因为ServiceB.methodB是新起一个事务，那么就是存在  两个不同的事务。如果ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚，ServiceB.methodB是不会回滚的。如果ServiceB.methodB失败回滚，  如果他抛出的异常被ServiceA.methodA捕获，ServiceA.methodA事务仍然可能提交。  5： PROPAGATION\_NOT\_SUPPORTED  当前不支持事务。比如ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED ，而ServiceB.methodB的事务级别是PROPAGATION\_NOT\_SUPPORTED ，  那么当执行到ServiceB.methodB时，ServiceA.methodA的事务挂起，而他以非事务的状态运行完，再继续ServiceA.methodA的事务。  6： PROPAGATION\_NEVER  不能在事务中运行。假设ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED， 而ServiceB.methodB的事务级别是PROPAGATION\_NEVER ，  那么ServiceB.methodB就要抛出异常了。  7： PROPAGATION\_NESTED  理解Nested的关键是savepoint。他与PROPAGATION\_REQUIRES\_NEW的区别是，PROPAGATION\_REQUIRES\_NEW另起一个事务，将会与他的父事务相互独立，  而Nested的事务和他的父事务是相依的，他的提交是要等和他的父事务一块提交的。也就是说，如果父事务最后回滚，他也要回滚的。  而Nested事务的好处是他有一个savepoint。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  ServiceA {  /\*\*  \* 事务属性配置为 PROPAGATION\_REQUIRED  \*/  void methodA() {  try {  //savepoint  ServiceB.methodB(); //PROPAGATION\_NESTED 级别  } catch (SomeException) {  // 执行其他业务, 如 ServiceC.methodC();  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  也就是说ServiceB.methodB失败回滚，那么ServiceA.methodA也会回滚到savepoint点上，ServiceA.methodA可以选择另外一个分支，比如  ServiceC.methodC，继续执行，来尝试完成自己的事务。  但是这个事务并没有在EJB标准中定义。  二、Isolation Level(事务隔离等级):  1、Serializable：最严格的级别，事务串行执行，资源消耗最大；  2、REPEATABLE READ：保证了一个事务不会修改已经由另一个事务读取但未提交（回滚）的数据。避免了“脏读取”和“不可重复读取”的情况，但是带来了更多的性能损失。  3、READ COMMITTED:大多数主流数据库的默认事务等级，保证了一个事务不会读到另一个并行事务已修改但未提交的数据，避免了“脏读取”。该级别适用于大多数系统。  4、Read Uncommitted：保证了读取过程中不会读取到非法数据。隔离级别在于处理多事务的并发问题。  我们知道并行可以提高数据库的吞吐量和效率，但是并不是所有的并发事务都可以并发运行，这需要查看数据库教材的可串行化条件判断了。  这里就不阐述。  我们首先说并发中可能发生的3中不讨人喜欢的事情  1： Dirty reads--读脏数据。也就是说，比如事务A的未提交（还依然缓存）的数据被事务B读走，如果事务A失败回滚，会导致事务B所读取的的数据是错误的。  2： non-repeatable reads--数据不可重复读。比如事务A中两处读取数据-total-的值。在第一读的时候，total是100，然后事务B就把total的数据改成 200，事务A再读一次，结果就发现，total竟然就变成200了，造成事务A数据混乱。  3： phantom reads--幻象读数据，这个和non-repeatable reads相似，也是同一个事务中多次读不一致的问题。但是non-repeatable reads的不一致是因为他所要取的数据集被改变了（比如total的数据），但是phantom reads所要读的数据的不一致却不是他所要读的数据集改变，而是他的条件数据集改变。比如Select account.id where account.name="ppgogo\*",第一次读去了6个符合条件的id，第二次读取的时候，由于事务b把一个帐号的名字由"dd"改成"ppgogo1"，结果取出来了7个数据。     |  |  |  |  | | --- | --- | --- | --- | |  | Dirty reads | non-repeatable reads | phantom reads | | Serializable | 不会 | 不会 | 不会 | | REPEATABLE READ | 不会 | 不会 | 会 | | READ COMMITTED | 不会 | 会 | 会 | | Read Uncommitted | 会 | 会 | 会 |     三、readOnly  事务属性中的readOnly标志表示对应的事务应该被最优化为只读事务。  这是一个最优化提示。在一些情况下，一些事务策略能够起到显著的最优化效果，例如在使用Object/Relational映射工具（如：Hibernate或TopLink）时避免dirty checking（试图“刷新”）。  四、Timeout  在事务属性中还有定义“timeout”值的选项，指定事务超时为几秒。在JTA中，这将被简单地传递到J2EE服务器的事务协调程序，并据此得到相应的解释      20110112  数据库提供了四种事务隔离级别, 不同的隔离级别采用不同的锁类开来实现.   在四种隔离级别中, Serializable的级别最高, Read Uncommited级别最低.   大多数数据库的默认隔离级别为: Read Commited,如Sql Server , Oracle.   少数数据库默认的隔离级别为Repeatable Read, 如MySQL InnoDB存储引擎   即使是最低的级别,也不会出现 第一类 丢失 更新问题 .    1. **脏读(事务没提交，提前读取)** ：脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。   2. **不可重复读(两次读的不一致)** ：是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。例如，一个编辑人员两次读取同一文档，但在两次读取之间，作者重写了该文档。当编辑人员第二次读取文档时，文档已更改。原始读取不可重复。如果只有在作者全部完成编写后编辑人员才可以读取文档，则可以避免该问题。  3.**幻读**: 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。例如，一个编辑人员更改作者提交的文档，但当生产部门将其更改内容合并到该文档的主复本时，发现作者已将未编辑的新材料添加到该文档中。如果在编辑人员和生产部门完成对原始文档的处理之前，任何人都不能将新材料添加到文档中，则可以避免该问题。  4.**第一类更新丢失(回滚丢失)** ：    当2个事务更新相同的数据源，如果第一个事务被提交，而另外一个事务却被撤销，那么会连同第一个事务所做的跟新也被撤销。也就是说第一个事务做的跟新丢失了。  5.**第二类更新丢失(覆盖丢失)** ：    第二类更新丢失实在实际应用中经常遇到的并发问题，他和不可重复读本质上是同一类并发问题，通常他被看做不可重复读的特例：当2个或这个多个事务查询同样的记录然后各自基于最初的查询结果更新该行时，会造成第二类丢失更新。因为每个事务都不知道不知道其他事务的存在，最后一个事务对记录做的修改将覆盖其他事务对该记录做的已提交的跟新...  补充 : 基于元数据的 Spring 声明性事务 :   Isolation 属性一共支持五种事务设置，具体介绍如下：   l          DEFAULT 使用数据库设置的隔离级别 ( 默认 ) ，由 DBA 默认的设置来决定隔离级别 .   l          READ\_UNCOMMITTED 会出现脏读、不可重复读、幻读 ( 隔离级别最低，并发性能高 )   l          READ\_COMMITTED  会出现不可重复读、幻读问题（锁定正在读取的行）   l          REPEATABLE\_READ 会出幻读（锁定所读取的所有行）   l          SERIALIZABLE 保证所有的情况不会发生（锁表）   不可重复读的重点是修改 :  同样的条件 ,   你读取过的数据 ,   再次读取出来发现值不一样了  幻读的重点在于新增或者删除  同样的条件 ,   第 1 次和第 2 次读出来的记录数不一样 |

# [Spring mvc 配置事务的注意点](http://blog.csdn.net/zxcvqwer19900720/article/details/20999623)

标签： [spring mvc](http://www.csdn.net/tag/spring%20mvc)[事务](http://www.csdn.net/tag/%e4%ba%8b%e5%8a%a1)

2014-03-11 10:58 6288人阅读 [评论](http://blog.csdn.net/zxcvqwer19900720/article/details/20999623#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zxcvqwer19900720/article/details/20999623#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

后台琐事（3） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

1.applicationContext.xml中，去掉对controller的扫描

  <context:component-scan base-package="com.trans">  
<context:exclude-filter type="annotation"  
expression="org.springframework.stereotype.Controller" />  
</context:component-scan>

2.\*\*-servlet.xml 中，去掉对 service的扫描，加入对controller的扫描

  <context:component-scan base-package="com.trans">  
  <context:include-filter type="annotation"  
expression="org.springframework.stereotype.Controller" />  
<context:exclude-filter type="annotation"  
expression="org.springframework.stereotype.Service" />  
</context:component-scan>

3.事务的配置只有在 applicationContext.xml中才会起作用，即

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

必须配置在 applicationContext.xml 中

4.用事务去控制的service，不能加try catch去捕获异常，否则不能被spring拦截到，事务就失效了。