**Spring事务管理**

**一、事务的基本原理**

[Spring事务](http://www.codeceo.com/article/spring-transactions.html)的**本质**其实就是**数据库对事务的支持**，没有数据库的事务支持，spring是无法提供事务功能的。对于纯JDBC操作数据库，想要用到事务，可以按照以下步骤进行：

1. 获取连接 Connection con = DriverManager.getConnection()
2. 开启事务con.setAutoCommit(true/false);
3. 执行CRUD //增删改查
4. 提交事务/回滚事务 con.commit() / con.rollback();
5. 关闭连接 conn.close();

使用Spring的事务管理功能后，我们**可以不再写步骤 2 和 4 的代码**，而是由Spirng 自动完成。 那么Spring是如何在我们书写的 CRUD 之前和之后开启事务和关闭事务的呢？解决这个问题，也就可以从整体上理解Spring的事务管理实现原理了。下面简单地介绍下，注解方式为例子

1. 配置文件开启注解驱动，在相关的类和方法上通过注解@Transactional标识。
2. spring 在启动的时候会去解析生成相关的bean，这时候会查看拥有相关注解的类和方法，并且为这些类和方法**生成代理**，并根据@Transaction的相关参数进行相关配置注入，这样就在代理中为我们把相关的事务处理掉了（开启正常提交事务，异常回滚事务）。
3. 真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

**二、Spring 事务的传播属性**

所谓spring事务的传播属性，就是定义：在多个事务同时存在的时候，spring应该如何处理这些事务的行为。这些属性在TransactionDefinition中定义，具体常量的解释见下表：

|  |  |
| --- | --- |
| 常量名称 | 常量解释 |
| 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，也是 Spring 默认的事务的传播。 |
| PROPAGATION\_REQUIRES\_NEW | 新建事务，如果当前存在事务，把当前事务挂起。新建的事务将和被挂起的事务没有任何关系，是两个独立的事务，外层事务失败回滚之后，不能回滚内层事务执行的结果，内层事务失败抛出异常，外层事务捕获，也可以不处理回滚操作 |
| 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\_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。 |
| PROPAGATION\_NESTED | 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按REQUIRED属性执行。它使用了一个单独的事务，这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器起效。 |

**三、数据库隔离级别**

|  |  |  |
| --- | --- | --- |
| 隔离级别 | 隔离级别的值 | 导致的问题 |
| Read-Uncommitted | 0 | 导致脏读 |
| Read-Committed | 1 | 避免脏读，允许不可重复读和幻读 |
| Repeatable-Read | 2 | 避免脏读，不可重复读，允许幻读 |
| Serializable | 3 | 串行化读，事务只能一个一个执行，避免了脏读、不可重复读、幻读。执行效率慢，使用时慎重 |

**脏读**：一事务对数据进行了增删改，但未提交，另一事务可以读取到未提交的数据。如果第一个事务这时候回滚了，那么第二个事务就读到了**脏数据**。

**不可重复读**：一个事务中发生了两次读操作，第一次读操作和第二次操作之间，另外一个事务对数据进行了修改，这时候两次读取的数据是不一致的。

**幻读**：第一个事务对一定范围的数据进行批量修改，第二个事务在这个范围增加一条数据，这时候第一个事务就会丢失对新增数据的修改。

**总结：**

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。

大多数的数据库默认隔离级别为 Read Commited，比如 SqlServer、Oracle

少数数据库默认隔离级别为：Repeatable Read 比如： MySQL InnoDB

**四、Spring中的隔离级别**

|  |  |
| --- | --- |
| 常量 | 解释 |
| ISOLATION\_DEFAULT | 这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。另外四个与 JDBC 的隔离级别相对应。 |
| ISOLATION\_READ\_UNCOMMITTED | 这是事务最低的隔离级别，它充许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。 |
| ISOLATION\_READ\_COMMITTED | 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。 |
| ISOLATION\_REPEATABLE\_READ | 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。 |
| ISOLATION\_SERIALIZABLE | 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。 |

**五、事务的嵌套**

通过上面的理论知识的铺垫，我们大致知道了数据库事务和spring事务的一些属性和特点，接下来我们通过分析一些嵌套事务的场景，来深入理解spring事务传播的机制。

假设外层事务 Service A 的 Method A() 调用 内层Service B 的 Method B()

**PROPAGATION\_REQUIRED(spring 默认)**

如果ServiceB.methodB() 的事务级别定义为 PROPAGATION\_REQUIRED，那么执行 ServiceA.methodA() 的时候spring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。

假如 ServiceB.methodB() 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。

**PROPAGATION\_REQUIRES\_NEW**

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，它才继续执行。

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(主要看B抛出的异常是不是A会回滚的异常)。

**PROPAGATION\_SUPPORTS**

假设ServiceB.methodB() 的事务级别为 PROPAGATION\_SUPPORTS，那么当执行到ServiceB.methodB()时，如果发现ServiceA.methodA()已经开启了一个事务，则加入当前的事务，如果发现ServiceA.methodA()没有开启事务，则自己也不开启事务。这种时候，内部方法的事务性完全依赖于最外层的事务。

**PROPAGATION\_NESTED**

现在的情况就变得比较复杂了, ServiceB.methodB() 的事务属性被配置为 PROPAGATION\_NESTED, 此时两者之间又将如何协作呢?  ServiceB#methodB 如果 rollback, 那么内部事务(即 ServiceB#methodB) 将回滚到它执行前的 SavePoint 而外部事务(即 ServiceA#methodA) 可以有以下两种处理方式:

a、捕获异常，执行异常分支逻辑

**void** **methodA**() {

**try** {

ServiceB.methodB();

} **catch** (SomeException) {

// 执行其他业务, 如 ServiceC.methodC();

}

}

这种方式也是嵌套事务最有价值的地方, 它起到了分支执行的效果, 如果 ServiceB.methodB 失败, 那么执行 ServiceC.methodC(), 而 ServiceB.methodB 已经回滚到它执行之前的 SavePoint, 所以不会产生脏数据(相当于此方法从未执行过), 这种特性可以用在某些特殊的业务中, 而 PROPAGATION\_REQUIRED 和 PROPAGATION\_REQUIRES\_NEW 都没有办法做到这一点。

b、 外部事务回滚/提交 代码不做任何修改, 那么如果内部事务(ServiceB#methodB) rollback, 那么首先 ServiceB.methodB 回滚到它执行之前的 SavePoint(在任何情况下都会如此), 外部事务(即 ServiceA#methodA) 将根据具体的配置决定自己是 commit 还是 rollback

另外三种事务传播属性基本用不到，在此不做分析。

**六、总结**

对于项目中需要使用到事务的地方，我建议开发者还是使用spring的TransactionCallback接口来实现事务，不要盲目使用spring事务注解，如果一定要使用注解，那么一定要对spring事务的传播机制和隔离级别有个详细的了解，否则很可能发生意想不到的效果。