链滴

Transaction 那点事儿

作者：jama

原文链接：https://ld246.com/article/1486717653636

来源网站：链滴

Transaction 也就是所谓的事务了，通俗理解就是一件事情。从小，父母就教育我们，做事情要始有终，不能半途而废。 事务也是这样，不能做一般就不做了，要么做完，要么就不做。也就是说，务必须是一个不可分割的整体，就像我们在化学课里学到的原子，原子是构成物质的最小单位。于是人们就归纳出事务的第一个特性：原子性（Atomicity）。我靠，一点都不神秘。
特别是在数据库领域，事务是一个非常重要的概念，除了原子性以外，它还有一个极其重要的特，那就是：一致性（Consistency）。也就是说，执行完数据库操作后，数据不被破坏。打个比方，如果从 A 账户转账到 B 账户，不可能因为 A 账户扣了钱，而 B 账户没有加钱吧如果出现了这类事情，您一定会非常气愤，什么 diao 银行啊！
当我们编写了一条 update 语句，提交到数据库的一刹那间，有可能别人也提交了一条 delete 句到数据库中。也许我们都是对同一条记录进行操作，可以想象，如果不稍加控制，就会出大麻烦来我们必须保证数据库操作之间是“隔离”的（线程之间有时也要做到隔离），彼此之间没有任何干扰这就是：隔离性（Isolation）。要想真正的做到操作之间完全没有任何干扰是难的，于是乎，每天上班打酱油的数据库专家们，开始动脑筋了，“我们要制定一个规范，让各个数库厂商都支持我们的规范！”，这个规范就是：**事****务隔离级别（Transaction Isolation Level。**能定义出这样牛逼的规范真的挺不容易的，其实说白了就四个级别：

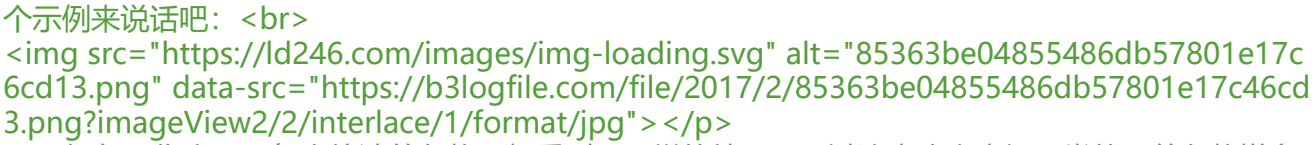
READ_UNCOMMITTED
READ_COMMITTED
REPEATABLE_READ
SERIALIZABLE

千万不要去翻译，那只是一个代号而已。从上往下，级别越来越高，并发性越来越差，安全性越越高，反之则反。
当我们执行一条 insert 语句后，数据库必须要保证有一条数据永久地存放在磁盘中，这个也算务的一条特性， 它就是：持久性（Durability）。
归纳一下，以上一共提到了事务的 4 条特性，把它们的英文单词首字母合起来就是：AID，这个就是传说中的“事务 ACID 特性”！
真的是非常牛逼的特性啊！这 4 条特性，是事务管理的基石，一定要透彻理解。此外还要明确，四个家伙当中，谁才是老大？
其实想想也就清楚了：原子性是基础，隔离性是手段，持久性是目的，真正的老大就是一致性。据不一致了，就相当于“江湖乱套了，流氓戴胸罩”。所以说，这三个小弟都是跟着“一致性”这个大混，为他全心全意服务。
这四个家伙当中，其实最难理解的反倒不是一致性，而是隔离性。因为它是保证一致性的重要手，是工具，使用它不能有半点差池，否则后果自负！怪不得数据库行业专家们都要来研究所谓的事务离级别了。其实，定义这四个级别就是为了解决数据在高并发下所产生的问题，那又有哪些问题呢？

Dirty Read（脏读）
Unrepeatable Read（不可重复读）
Phantom Read（幻读）

首先看看“脏读”，看到“脏”这个字，我就想到了恶心、肮脏。数据怎么可能脏呢？其实也就我们经常说的“垃圾数据”了。比如说，有两个事务，它们在并发执行（也就是竞争）。看看以下这表格，您一定会明白我在说什么：

余额应该为 1500 元才对！请看 T5 时间点，事务 A 此时查询余额为 0 元，这个数据就是脏数据它是事务 B 造成的，明显事务没有进行隔离，渗过来了，乱套了。
所以脏读这件事情是非常要不得的，一定要解决掉！让事务之间隔离起来才是硬道理。
那第 2 条，不可重复读又怎么解释呢？还是用类似的例子来说明：

png?imageView2/2/interlace/1/format/jpg">
事务 A 其实除了查询了两次以外，其他什么事情都没有做，结果钱就从 1000 变成 0 了，这就重复读了。可想而知，这是别人干的，不是我干的。其实这样也是合理的，毕竟事务 B 提交了事务，据库将结果进行了持久化，所以事务 A 再次读取自然就发生了变化。
这种现象基本上是可以理解的，但在有些变态的场景下却是不允许的。毕竟这种现象也是事务之没有隔离所造成的，但我们对于这种问题，似乎可以忽略。
最后一条，幻读。我去！Phantom 这个单词不就是“幽灵、鬼魂”吗？刚看到这个单词时，真把我的小弟弟都给惊呆了。怪不得这里要翻译成“幻读”了，总不能翻译成“幽灵读”、“鬼魂读”。其实意义就是鬼在读，不是人在读，或者说搞不清楚为什么，它就变了，很晕，真的很晕。还是用个示例来说话吧：


银行工作人员，每次统计总存款，都看到不一样的结果。不过这也确实也挺正常的，总存款增多，肯定是这个时候有人在存钱。但是如果银行系统真的这样设计，那算是玩完了。这同样也是事务没隔离所造成的，但对于大多数应用系统而言，这似乎也是正常的，可以理解，也是允许的。银行里那恶心的那些系统，要求非常严密，统计的时候，甚至会将所有的其他操作给隔离开，这种隔离级别就非常高了（估计要到 SERIALIZABLE 级别了）。
归纳一下，以上提到了事务并发所引起的跟读取数据有关的问题，各用一句话来描述下：

脏读：事务 A 读取了事务 B 未提交的数据，并在这个基础上又做了其他操作。
不可重复读：事务 A 读取了事务 B** 已提交的更改数据。**
幻读：事务 A 读取了事务 B 已提交****的新增数据。

第一条是坚决抵制的，后两条在大多数情况下可不作考虑。
这就是为什么必须要有事务隔离级别这个东西了，它就像一面墙一样，隔离不同的事务。看下面个表格，您就清楚了不同的事务隔离级别能处理怎样的事务并发问题：


根据您的实际需求，再参考这张表，最后确定事务隔离级别，应该不再是一件难事了。
JDBC 也提供了这四类事务隔离级别，但默认事务隔离级别对不同数据库产品而言，却是不一样。我们熟知的 MySQL 数据库的默认事务隔离级别就是 READ_COMMITTED，Oracle、SQL Server DB2 等都有有自己的默认值。我认为 READ_COMMITTED 已经可以解决绝大多数问题了，其他的就体情况具体分析吧。
若对其他数据库的默认事务隔离级别不太清楚，可以使用以下代码来获取：
DatabaseMetaData meta = DBUtil.getDataSource().getConnection().getMetaData(); int defaultsolation = meta.getDefaultTransactionIsolation();
提示：在 java.sql.Connection 类中可查看所有的隔离级别。
我们知道 JDBC 只是连接 Java 程序与数据库的桥梁而已，那么数据库又是怎样隔离事务的呢？实它就是“锁”这个东西。当插入数据时，就锁定表，这叫“锁表”；当更新数据时，就锁定行，这“锁行”。当然这个已经超出了我们今天讨论的范围，所以还是留点空间给我们的 DBA 同学吧，免他没啥好写的了。
除了 JDBC 给我们提供的事务隔离级别这种解决方案以外，还有哪些解决方案可以完善事务管理能呢？
不妨看看 Spring 的解决方案吧，其实它是对 JDBC 的一个补充或扩展。它提供了一个非常重要功能，就是：事务传播行为（Transaction Propagation Behavior）。
确实够牛逼的，Spring 一下子就提供了 7 种事务传播行为，这 7 种行为一出现，真的是亮瞎了的狗眼！

PROPAGATION_REQUIRED
RROPAGATION_REQUIRES_NEW

PROPAGATION_NESTED
PROPAGATION_SUPPORTS
PROPAGATION_NOT_SUPPORTED
PROPAGATION_NEVER
PROPAGATION_MANDATORY

看了 Spring 参考手册之后，更是晕了，这到底是在干嘛？
首先要明确的是，事务是从哪里来？传播到哪里去？答案是，从方法 A 传播到方法 B。Spring 决的只是方法之间的事务传播，那情况就多了，比如：

方法 A 有事务，方法 B 也有事务。
方法 A 有事务，方法 B 没有事务。
方法 A 没有事务，方法 B 有事务。
方法 A 没有事务，方法 B 也没有事务。

这样就是 4 种了，还有 3 种特殊情况。还是用我的 Style 给大家做一个分析吧：
假设事务从方法 A 传播到方法 B，您需要面对方法 B，问自己一个问题：
方法 A 有事务吗？

如果没有，就新建一个事务；如果有，就加入当前事务。这就是 PROPAGATION_REQUIRED，也是 Spring 提供的默认事务传播行为，适合绝大多数情况。
如果没有，就新建一个事务；如果有，就将当前事务挂起。这就是 RROPAGATION_REQUIRES_EW，意思就是创建了一个新事务，它和原来的事务没有任何关系了。
如果没有，就新建一个事务；如果有，就在当前事务中嵌套其他事务。这就是 PROPAGATION_ESTED，也就是传说中的“嵌套事务”了，所嵌套的子事务与主事务之间是有关联的（当主事务提交回滚，子事务也会提交或回滚）。
如果没有，就以非事务方式执行；如果有，就使用当前事务。这就是 PROPAGATION_SUPPORT，这种方式非常随意，没有就没有，有就有，有点无所谓的态度，反正我是支持你的。
如果没有，就以非事务方式执行；如果有，就将当前事务挂起。这就是 PROPAGATION_NOT_SPPORTED，这种方式非常强硬，没有就没有，有我也不支持你，把你挂起来，不鸟你。
如果没有，就以非事务方式执行；如果有，就抛出异常。这就是 PROPAGATION_NEVER，这种式更猛，没有就没有，有了反而报错，确实够牛的，它说：我从不支持事务！
如果没有，就抛出异常；如果有，就使用当前事务。这就是 PROPAGATION_MANDATORY，种方式可以说是牛逼中的牛逼了，没有事务直接就报错，确实够狠的，它说：我必须要有事务！

看到我上面这段解释，小伙伴们是否已经感受到，被打通任督二脉的感觉？多读几遍，体会一下就是您自己的东西了。
需要注意的是 PROPAGATION_NESTED，不要被它的名字所欺骗，Nested（嵌套），所以凡在类似方法 A 调用方法 B 的时候，在方法 B 上使用了这种事务传播行为，如果您真的那样做了，那就错了。因为您错误地以为 PROPAGATION_NESTED 就是为方法嵌套调用而准备的，其实默认的 POPAGATION_REQUIRED 就可以帮助您，做您想要做的事情了。
Spring 给我们带来了事务传播行为，这确实是一个非常强大而又实用的功能。除此以外，也提了一些小的附加功能，比如：

事务超时（Transaction Timeout）：为了解决事务时间太长，消耗太多资源，所以故意给事务设置一个最大时常，如果超过了，就回滚事务。
只读事务（Readonly Transaction）：为了忽略那些不需要事务的方法，如读取数据，这样可以有效地提高一些性能。

最后，推荐大家使用 Spring 的注解式事务配置，而放弃 XML 式事务配置。因为注解实在是太雅了，当然这一切都取决于您自身的情况了。
在 Spring 配置文件中使用：
<pre><code class="highlight-chroma">...

...
</code></pre>
在需要事务的方法上使用：
<pre><code class="highlight-chroma">@Transactional
public void xxx() {
    ...
}
</code></pre>
可在 @Transactional 注解中设置：事务隔离级别、事务传播行为、事务超时时间、是否只读事。
简直是太完美了，太优雅了！
