JAVA资格认证:J2EE事务并发控制策略总结Java认证考试 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/559/2021_2022_JAVA_E8_B5_84_E6_A0_BC_c104_559564.htm 本文结合Hibernate以及JPA标准，对J2EE当前持久层设计所遇到的几个问题进行总结： 事务并发访问控制策略 当前J2EE项目中，面临的一个共同问题就是如果控制事务的并发访问，虽然有些持久层框架已经为我们做了很多工作，但是理解原理，对于我们开发来说还是很有用处的。 事务并发访问主要可以分为两类，分别是同一个系统事务和跨事务访问的并发访问控制，其中同一个系统事务可以采取乐观锁以及悲观锁策略，而跨多个系统事务时则需要乐观离线锁和悲观离线锁。在讨论这四种并发访问控制策略之前，先需要明确一下数据库事务隔离级别的问题，ANSI标准规定了四个数据库事务隔离级别，它们分别是：读取未提交（Read Uncommitted) 这是最低的事务隔离级别，读事务不会阻塞读事务和写事务，写事务也不会阻塞读事务，但是会阻塞写事务。这样造成的一个结果就是当一个写事务没有提交的时候，读事务照样可以读取，那么造成了脏读的现象。 读取已提交(Read Committed) 采用此种隔离界别的时候，写事务就会阻塞读事务和写事务，但是读事务不会阻塞读事务和写事务，这样因为写事务会阻塞读取事务，那么从而读取事务就不能读到脏数据，但是因为读事务不会阻塞其它的事务，这样还是会造成不可重复读的问题。 可重复读（Repeatable Read) 采用此种隔离级别，读事务会阻塞写事务，但是读事务不会阻塞读事务，但是写事务会阻塞写事务和读事务。因为读事务阻塞了写事务，这样以来就不会造成不

可重复读的问题，但是这样还是不能避免幻影读问题。 序列化（serializable) 此种隔离级别是最严格的隔离级别，如果设置成这个级别，那么就不会出现以上所有的问题（脏读，不可重复读，幻影读）。但是这样以来会极大的影响到我们系统的性能，因此我们应该避免设置成为这种隔离级别，相反的，我们应该采用较低的隔离界别，然后再采用并发控制策略来进行事务的并发访问控制）。 其实我们也可以把事务隔离级别设置为serializable，这样就不需要采用并发控制策略了，数据库就会为我们做好一切并发控制，但是这样以来会严重影响我们系统的伸缩性和性能，所以在实践中，我们一般采用读取已提交或者更低的事务隔离级别，配合各种并发访问控制策略来达到并发事务控制的目的。下面总结一下常用的控制策略：1 乐观锁 乐观锁是在同一个数据库事务中我们常采取的策略，因为它能使得我们的系统保持高的性能的情况下，提高很好的并发访问控制。乐观锁，顾名思义就是保持一种乐观的态度，我们认为系统中的事务并发更新不会很频繁，即使冲突了也没事，大不了重新再来一次。它的基本思想就是每次提交一个事务更新时，我们想看看要修改的东西从上次读取以后有没有被其它事务修改过，如果修改过，那么更新就会失败,。 最后我们需要明确一个问题，因为乐观锁其实并不会锁定任何记录，所以如果我们数据库的事务隔离级别设置为读取已提交或者更低的隔离界别，那么是不能避免不可重复读问题的（因为此时读事务不会阻塞其它事务），所以采用乐观锁的时候，系统应该要容许不可重复读问题的出现。 了解了乐观锁的概念以后，那么当前我们系统中又是如何来使用这种策略的呢？一般可以采用以下三种方法

：版本(Version)字段：在我们的实体中增加一个版本控制字段，每次事务更新后就将版本字段的值加1. 时间戳（timestamps)：采取这种策略后，当每次要提交更新的时候就会将系统当前时间和实体加载时的时间进行比较，如果不一致，那么就报告乐观锁失败，从而回滚事务或者重新尝试提交。采用时间戳有一些不足，比如在集群环境下，每个节点的时间同步也许会成问题，并且如果并发事务间隔时间小于当前平台最小的时钟单位，那么就会发生覆盖前一个事务结果的问题。因此一般采用版本字段比较好。 基于所有属性进行检测：采用这种策略的时候，需要比较每个字段在读取以后有没有被修改过，所以这种策略实现起来比较麻烦，要求对每个属性都进行比较，如果采用hiernate的话，因为Hibernate在一级缓存中可以进行脏检测，那么可以判断哪些字段被修改过，从而动态的生成sql语句进行更新。 下面再总结一下如何在JDBC和Hibernate中使用乐观锁： JDBC中使用乐观锁：如果我们采用JDBC来实现持久层的话，那么就可以采用以上将的三种支持乐观锁的策略，在实体中增加一个version字段或者一个Date字段，也可以采用基于所有属性的策略，下面就采用version字段来做一演示： 假如系统中有一个Account的实体类，我们在Account中多加一个version字段，那么我们JDBC Sql语句将如下写： Select a.version....from Account as a where (where condition..) Update Account set version = version 1.....(another field) where version =?...(another contidition) 这样以来我们就可以通过更新结果的行数来进行判断，如果更新结果的行数为0，那么说明实体从加载以来已经被其它事务更改了，所以就抛出自定义的乐观锁定异常（

或者也可以采用Spring封装的异常体系）。具体实例如下：
..... int rowsUpdated = statement.executeUpdate(sql).
If(rowsUpdated= =0){ throws new
OptimisticLockingFailureException(). } ..... 在使用JDBC API的情况下，我们需要在每个0update语句中，都要进行版本字段的更新以及判断，因此如果稍不小心就会出现版本字段没有更新的问题，相反当前的 ORM框架却为我们做好了一切，我们仅仅需要做的就是在每个实体中都增加version或者是Date字段。 Hibernate中使用乐观锁：如果我们采用Hibernate做为持久层的框架，那么实现乐观锁将变得非常容易，因为框架会帮我们生成相应的sql语句，不仅减少了开发人员的负担，而且不容易出错。下面同样采用version字段的方式来总结一下： 同样假如系统中有一个Account的实体类，我们在Account中多加一个version字段， public class Account{ Long id . ..... @Version //也可以采用XML文件进行配置 Int version ..... } 这样以来每次我们提交事务时，hibernate内部会生成相应的SQL语句将版本字段加1，并且进行相应的版本检测，如果检测到并发乐观锁定异常，那么就抛出StaleObjectStateException. 2 悲观锁 所谓悲观锁，顾名思义就是采用一种悲观的态度来对待事务并发问题，我们认为系统中的并发更新会非常频繁，并且事务失败了以后重来的开销很大，这样以来，我们就需要采用真正意义上的锁来进行实现。悲观锁的基本思想就是每次一个事务读取某一条记录后，就会把这条记录锁住，这样其它的事务要想更新，必须等以前的事务提交或者回滚解除锁。 最后我们还是需要明确一个问题，假如我们数据库事务的隔离级别设置为读取已提交

或者更低，那么通过悲观锁，我们控制了不可重复读的问题，但是不能避免幻影读的问题（因为要想避免我们就需要设置数据库隔离级别为Serializable,而一般情况下我们都会采取读取已提交或者更低隔离级别，并配合乐观或者悲观锁来实现并发控制，所以幻影读问题是不能避免的，如果想避免幻影读问题，那么你只能依靠数据库的serializable隔离级别（幸运的是幻影读问题一般情况下不严重）。 下面就分别以JDBC和Hibernate来总结一下： JDBC中使用悲观锁：在JDBC中使用悲观锁，需要使用0select for 0update语句，假如我们系统中有一个Account的类，我们可以采用如下的方式来进行： Select * from Account where ...(where condition).. for 0update. 当使用了for 0update语句后，每次在读取或者加载一条记录的时候，都会锁住被加载的记录，那么当其他事务如果要更新或者是加载此条记录就会因为不能获得锁而阻塞，这样就避免了不可重复读以及脏读的问题，但是其他事务还是可以插入和删除记录，这样也许同一个事务中的两次读取会得到不同的结果集，但是这不是悲观锁锁造成的问题，这是我们数据库隔离级别所造成的问题。 最后还需要注意的一点就是每个冲突的事务中，我们必须使用0select for 0update 语句来进行数据库的访问，如果一些事务没有使用0select for 0update语句，那么就会很容易造成错误，这也是采用JDBC进行悲观控制的缺点。 Hibernate中使用悲观锁：相比于JDBC使用悲观锁来说，在Hibernate中使用悲观锁将会容易很多，因为Hibernate有API让我们来调用，从而避免直接写SQL语句。下面就Hibernate使用悲观锁做一总结： 首先先要明确一下Hibernate中支持悲观锁的两种模式LockMode.UPGRADE

以LockMode.UPGRADE_NO_WAIT.(PS:在JPA中，对应的锁模式是LockModeType.Read，这与Hibernate是不一样的呵呵)假如我们系统中有一个Account的类，那么具体的操作可以像这样： ..... session.lock(account, LockMode.UPGRADE). ..... 或者也可以采用如下方式来加载对象：session.get(Account.class,identity,LockMode.UPGRADE). 这样以来当加载对象时，hibernate内部会生成相应的0select for 0update语句来加载对象，从而锁定对应的记录，避免其它事务并发更新。 以上两种策略都是针对同一个事务而言的，如果我们要实现跨多个事务的并发控制就要采用其它两种并发控制策略了，下面做一总结： C 与java是两种完全不同风格的东西，C 是由程序员创造的，由程序员完善的，然后才出的标准的，也就是说C 的标准完全落后与C 的发展。java恰好相反，它是先有标准（可能还没有实现），然后后有的实现，而且它是由公司主导开发的，虽然现在开源了，但是标准并不是谁都能定的。这就造就了C 是百花齐放，博大精深，很少有人敢说自己C 很厉害。java却是另外的一种感觉，一切都规定好了，你只需要按照规定去做，符合标准才可以的。所以C 是那种既可以做的堂堂正正，博大精深（比如标准库），又可以实现的匪夷所思，天马行空（写 Boost库的人太牛了）。java不行，java要求如此只能如此，不能越雷池一步。