**事务，隔离级别，锁**

1. 事务类型（这里JavaEE与Spring的定义基本相同）

Required：如果Context中有事务就加入，没有就自己创建一个。（最常用设置）

Mandatory：永远加入一个事务。如果当前Context没有事务，抛出异常。（那些不打算自己负责rollback事务的方法，必须加入到别人的事务，由别人来控制rollback）

RequiresNew：永远新建一个事务。（那些不管别人如何，自己必须提交事务的方法，比如审计信息是一定要写的）

Supports：如果有事务就加入，没有就算了。永远不会创建新事务。（一般用于只读方法，不会主动创建事务，但如果当前有事务就加入，以读到事务中未提交的数据）

NotSupported：永远不使用事务，如果当前有事务，挂起事务。（那些有可能抛异常并不影响全局的方法）

Never：不能在有当前事务的情况下调用本方法。

可见，Required是默认的设置，Supports是只读方法的最佳选择。

1. 数据库并发操作的一些常见问题

情况1（DIRTY READ）、

Mary的原工资为1000， 财务人员将Mary的工资改为了8000(但未提交事务)。

Mary读取自己的工资，发现自己的工资变为了8000！

而财务发现操作有误，回滚了事务，Mary的工资又变为了1000，像这样，Mary记取的工资数8000是一个脏数据。

情况2（UNREPEATABLE READ）、

A 开始一个事务TA,然后从数据库中查找X < 1的记录，结果找到一条记录 X = 0。

在A提交事务TA前，B开始了一个新的事务TB, B先从数据库中读出X = 0, 然后将X设置为 X = X + 1,然后将X保存回数据库，最后提交事务 TB，此时数据库中X = 1。

A重新在数据库中查找X < 1的记录，结果发现没有任何纪录满足条件！这是因为在事务TA提交前，有一个新的事务TB更新了X的值，它导致了在同一个事务TA中的读操作时不可重复的。

情况3（PHANTOM）、

开始一个事务TA（transaction），然后从数据库中查找X >= 0的记录，结果找到一条记录 [X = 0]。

在 A 提交事务（commit）前，B开始了一个事务TB，向数据库中插入一条记录X = 2,然后提交事务TB。

A又一次从数据库中查找X >= 0的记录，结果找到两条记录[X=0, X=2]！这就导致了TA的第二次查询出现了幻象。

情况4（第一类更新丢失--回滚丢失）、

当两个事物更新相同的数据源，如果第一个事务被提交，而另一个事务却被撤消，那么会连同第一个事务所做的更新也被撤消。

情况5（第二类更新丢失—覆盖丢失）、

第二类更新丢失在实际应用中经常遇到的并发问题，它和不可重复读本质上是同类问题，通常被看作不可重复读的特例。当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，会造成第二类更新丢失。因为每个事务都不知道其它事务的存在，最后一个事务对记录做的修改将覆盖其它事务对该记录做的已提交更新。

1. JDBC中的隔离级别(事务隔离级别)

为了避免出现上面所说的种种问题，需要JDBC提供了一系列的隔离级别来帮助我们进行并发控制：

**1． READ UNCOMMITED（简称UR）**

这个隔离级别是最弱的隔离级别，事实上他没有任何隔离措施，也不能解决上述的任何问题。因此要采用这个隔离级别的前提是您实现知道一定不会有并发操作出现。对于现实的应用程序来说，这个隔离级别往往是不够的。但是这个隔离级别也有它的好处，那就是它在性能上是最优的，底层的事务系统不需要在任何共享数据上加锁。

**2． READ COMMITTED（简称CS）**

在这个隔离级别上，我们可以避免脏读(DIRTY READ)的问题，它能保证应用程序不会读到已经更新但还没有提交的数据。使用这个隔离级别的应用程序可能是一个报表程序，它从数据库中读出数据，然后呈现给用户。在这个隔离级别上，它的底层实现需在共享数据上加一定的锁，所以从性能上来说，要比READ UNCOMMITTED略慢一点。注意：对于READ COMITTED这个隔离级别，在执行SQL SELECT操作期间数据库会锁住读取得记录，但是一旦读取操作完成，数据库记录上的锁就会被释放。

**3． REPEATABLE READ（简称RS）**

在这个隔离级别上，我们可以避免脏读(DIRTY READ)的问题和(UNREPEATABLE READ)的问题，它保证在一个事务中，无论我们什么时间执行查询操作，我们读到的数据不但肯定是已经提交的数据，而且任何两次查询操作的结果都是相同的。从底层实现来看，这个隔离级别需要给共享数据加上一种比在READ COMMITTED级别上更强的锁，因此从性能上来说，它要比READ COMMITTED略慢。

重点提示:

隔离事务引用的每一行都被锁定；但是，在该隔离级别下，只锁定隔离事务实际检索和／或修改的行。因此，如果一个事务扫描了 1000 行但只检索 10 行，则只有被检索到的 10 行（而不是所扫描的 1000 行）被锁定。

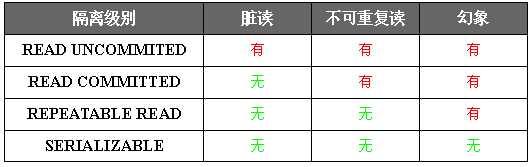
**4． SERIALIZABLE（简称RR）**

这个隔离级别是最强的隔离级别，它除了能避免脏读(DIRTY READ)的问题和(UNREPEATABLE READ)的问题外，还能保证在一个事务中，不会有新的数据被插入或已有的数据被删除，这也就解决了幻象问题(PHANTOM)。作为最强的隔离级别，它的底层实现对共享数据加的锁也是最强的锁，它把所有的操作都序列化了下来，这样它虽然能解决上面说的所有问题，但是在性能上也是最差的一个。

重点提示：

使用可重复的读隔离级别的事务可以多次检索同一行集，并可以对它们执行任意次操作，直到由提交或回滚操作终止事务；不允许其它事务执行插入、更新或删除操作，因为这些操作会在隔离事务存在期间影响正在被使用的行集。为了确保在“可重复的读”隔离级别下运行的事务所访问的数据不会受其它事务的负面影响，所以锁定了该隔离事务所引用的每个行 — 而不是仅锁定被实际检索和／或修改的那些行。因此，如果一个事务扫描了 1000 行但只检索 10 行，则所扫描的 1000 行（而不仅是被检索的 10 行）都会被锁定。

下面这张表格总结了各种隔离级别所解决的问题：



在这些隔离级别的底层实现上，我们是通过数据库所提供的锁机制来完成的。

1. 数据库锁的分类

1． 共享锁（SHARED）

当一个事务试图用SQL SELECT读取数据库中的记录时，它会获取一个共享锁，如果一个事务获取了一个共享锁，则它会阻止其他事务在同样的数据库记录上获取排斥锁（EXCLUSIVE）。但是它不会阻止其他事务在同样的数据库记录上获取共享锁（SHARED）或者更新锁（UPDATE）。

2． 更新锁（UPDATE）

当一个事务试图用SQL SELECT ... FOR UPDATE来读取数据库记录时，它会获取一个更新锁，如果一个事务获取了更新锁，那么它会阻止其他事务在同样的数据库记录上获取更新锁或者排斥锁，但是它不阻止其他事务在同样的数据库记录上获取共享锁。如果在获取更新锁之前，该事务已经有一个共享锁了，那么这个共享锁将被提升为一个更新锁，而不是重新分配一个新的更新锁给这个事务。

3． 排斥锁（EXCLUSIVE）。当一个事务试图增加，修改或删除数据库记录 （SQL UPDATE, DELETE, INSERT）时它会获取一个排斥锁，如果一个事务获取了排斥锁，那么它会阻止其他事务在同样的数据库记录上获取任何锁。如果在获取排斥锁之前，该事务已近有一个共享锁或者更新锁了，那么之前的获取的锁将被提升为一个排斥锁，而不是重新分配一个新的排斥锁给这个事务。

1. 正确使用锁和隔离级别

情况1（第二类更新丢失）、

假设有两个并发事务TA和TB，他们分别从数据库中取出一条记录然后再对该记录进行一个更新操作，那么如果TA和TB的隔离级别为READ COMIMITTED（CS），那么就会出现下面的情况：

1． TA用 SQL SELECT从数据库中读取了一条记录X = 0， 我们会获得一个共享锁L1，这保证在读取数据的过程中，读取的数据一定不会是未提交的事务中的数据，但是注意，在介绍事务隔离级别时我们讲到，对于READ COMMITTED，当读取操作完成后，锁就会被释放，所以当这一步完成后我们的共享锁L1就被释放了。

2． TB也用SQL SELECT从数据库中读取X=0，并获取锁L2，读操作完成后，L2也同样被释放。

3． TA用SQL UPDATE去更新X=1，由于L2已经被释放，所以更新操作不会被阻止。

4． TA用Commit提交事务，数据库被成功更新。

5． TB用SQL UPDATE去更新X=2，由于L1已经被释放，所以更新操作不会被阻止。

6． TB用Commit提交事务，数据库被成功更新。

这里就出现了一个问题TA的更新操作被TB所覆盖。

情况2（死锁）、

如果我们用隔离级别REPEATABLE READ（RS） 或者SERIALIZABLE（RR），并且在读取操作时使用共享锁，我们来看看会出现什么情况：

1． TA用SQL SELECT从数据库中读取一条记录X = 0，我们获得一个共享锁L1，它能保证我们读取的数据一定是已经提交的数据，同时由于隔离级别是REPEATABLE READ或SERIALIZABLE，所以这一步结束后，L1不会被释放。

2． TB 用SQL SELECT从数据库中读取X=0，并获得共享锁L2，应为L2是共享锁，所以这一步操作不会被L1阻止，同样L2也不会被释放。

3． TA 用SQL UPDATE试图去更新X=1，由于是UPDATE操作，所以数据库会尝试将L1提升为排斥锁，但是由于L2还没有被释放，所以TA不得不在这里等待TB释放L2。

4． TB用SQL UPDATE试图去更新X=2，由于是UPDATE操作，所以数据库会尝试将L2提升为排斥锁，但是由于L1还没有被释放，所以TB也不得不等待TA释放L1。

可怕的死锁问题出现了！

解决办法：

1. TA 用SQL SELECT ... FOR UPDATE从数据库中读取X = 0， 我们获得一个更新锁L1同时由于隔离级别是REPEATABLE READ（RS）或SERIALIZABLE（RR），所以这一步结束后，L1不会被释放。

2. TB 用SQL SELECT ... FOR UPDATE试图从数据库中读取X，由于L1没有被释放，而SELECT ... FOR UPDATE要求一个更新锁，所以这一步会被L1阻止。

3. TA用SQL UPDATE去更新X=1，由于没有其他锁存在，所以这一步操作不会被阻止，数据被成功更新。

4. TA用Commit提交事务，L1被释放。

重点提示：

对于隔离级别和锁的使用除了上面所说的种种问题外，我们还需要注意： SQL SELECT ... FOR UPDATE并不是对所有SQL语句都有效的，简单来说就是SELECT ... FOR UPDATE对只读性的查询是不适用的，比如需要有两表或多表连接 (JOIN) 的查询，有ORDER BY，GROUP BY，DISTINCT的查询语句等。另外锁和隔离级别的详细的语义可能也会应为具体数据库的不同而有所差别。

1. 加锁的策略

设想我们的应用程序对数据的逻辑是先读取数据，然后经过计算来决定是否对读取出来的数据进行更新操作。如果在实际运行时，绝大多数情况下是只做读取不做更新，只是偶尔才会有更新操作，在这种情况下，如果读取数据之后，提交事务之前，我们还有一些耗时的操作要完成，那么从性能的角度来看，如果采用隔离级别REPEATABLE READ 或者SERIALIZABLE，我们的性能是比较低的，因为我们为了保证偶尔的更新操作不会出现上面所说的并发操作的种种问题，对大多数的读取操作也加上了级别比较高的锁。为了解决这种问题我们有两种加锁的策略：

1、悲观策略（PESSIMISTIC），在这种策略下，我们在事务一开始时就获取一个锁，并一直保持这个锁，一直到事务结束位置。使用这种策略的好处是我们可以尽量地避免可能出现的并发操作的问题，缺点是对于那种大多数情况下是读，偶尔还有更新操作的情况，性能比较差。

2、乐观策略（OPTIMISTIC），在这种情况下，事务开始后，数据读取操作完成后，获取的锁就会被释放。等到更新操作时再获取新的排斥锁。使用这种策略，对于上面所说的大多数情况下读取，偶尔有更新的操作的情况，它能大大提高性能，应为它使得并发性更好。缺点是如果出现并发的更新操作，就有可能出现更新被覆盖（第二类更新丢失）的问题。

1. 友情提示

了解脏读，不可重复读，可重复读，幻读，共享锁，更新锁，排斥锁的概念。了解隔离级别中锁的使用。