[事务并发、事务隔离级别](http://blog.csdn.net/ocean1010/article/details/6548771)

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

2011-06-16 16:04 14307人阅读 [评论](http://blog.csdn.net/ocean1010/article/details/6548771#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ocean1010/article/details/6548771#report)

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

Sybase（5） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg SQL Server（20） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg mysql（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg Oracle（34） 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。

并发问题可归纳为以下几类:

A.丢失更新：撤销一个事务时，把其他事务已提交的更新数据覆盖（A和B事务并发执行，A事务执行更新后，提交；B事务在A事务更新后，B事务结束前也做了对该行数据的更新操作，然后回滚，则两次更新操作都丢失了）。  
  
B.脏读：一个事务读到另一个事务未提交的更新数据（A和B事务并发执行，B事务执行更新后，A事务查询B事务没有提交的数据，B事务回滚，则A事务得到的数据不是数据库中的真实数据。也就是脏数据，即和数据库中不一致的数据）。  
  
C.不可重复读：一个事务读到另一个事务已提交的更新数据（A和B事务并发执行，A事务查询数据，然后B事务更新该数据，A再次查询该数据时，发现该数据变化了）。  
  
D. 覆盖更新：这是不可重复读中的特例，一个事务覆盖另一个事务已提交的更新数据（即A事务更新数据，然后B事务更新该数据，A事务查询发现自己更新的数据变了）。

E.虚读（**幻读**）：一个事务读到另一个事务已提交的新插入的数据（A和B事务并发执行，A事务查询数据，B事务插入或者删除数据，A事务再次查询发现结果集中有以前没有的数据或者以前有的数据消失了）。  
  
数据库系统提供了四种事务隔离级别供用户选择：  
  
A.Serializable（串行化）：一个事务在执行过程中完全看不到其他事务对数据库所做的更新（事务执行的时候不允许别的事务并发执行。事务串行化执行，事务只能一个接着一个地执行，而不能并发执行。）。  
  
B.Repeatable Read（可重复读）：一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，但是不能看到其他其他事务对已有记录的更新。  
  
C.Read Commited（读已提交数据）：一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，而且能看到其他事务已经提交的对已有记录的更新。  
  
D.Read Uncommitted（读未提交数据）：一个事务在执行过程中可以看到其他事务没有提交的新插入的记录，而且能看到其他事务没有提交的对已有记录的更新。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 丢失更新 | 脏读 | 非重复读 | 覆盖更新 | 幻像读 |
| 未提交读 | Y | Y | Y | Y | Y |
| 已提交读 | N | N | Y | Y | Y |
| 可重复读 | N | N | N | N | Y |
| 串行化 | N | N | N | N | N |

**隔离级别**

数据库系统有四个隔离级别（大多数数据库默认级别为read commited）。对数据库使用何种隔离级别要审慎分析，因为

1. 维护一个最高的隔离级别虽然会防止数据的出错，但是却导致了并行度的损失，以及导致死锁出现的可能性增加。

2. 然而，降低隔离级别，却会引起一些难以发现的bug。

**SERIALIZABLE（序列化）**

添加范围锁（比如表锁，页锁等，关于range lock，我也没有很深入的研究），直到transaction A结束。以此阻止其它transaction B对此范围内的insert，update等操作。

幻读，脏读，不可重复读等问题都不会发生。

**REPEATABLE READ（可重复读）**

对于读出的记录，添加共享锁直到transaction A结束。其它transaction B对这个记录的试图修改会一直等待直到transaction A结束。

可能发生的问题：当执行一个范围查询时，可能会发生幻读。

**READ COMMITTED（提交读）**

在transaction A中读取数据时对记录添加共享锁，但读取结束立即释放。其它transaction B对这个记录的试图修改会一直等待直到A中的读取过程结束，而不需要整个transaction A的结束。所以，在transaction A的不同阶段对同一记录的读取结果可能是不同的。

可能发生的问题：不可重复读。

**READ UNCOMMITTED(未提交读)**

不添加共享锁。所以其它transaction B可以在transaction A对记录的读取过程中修改同一记录，可能会导致A读取的数据是一个被破坏的或者说不完整不正确的数据。

另外，在transaction A中可以读取到transaction B（未提交）中修改的数据。比如transaction B对R记录修改了，但未提交。此时，在transaction A中读取R记录，读出的是被B修改过的数据。

可能发生的问题：脏读。

**问题**

我们看到，当执行不同的隔离级别时，可能会发生各种各样不同的问题。下面对它们进行总结并举例说明。

**幻读**

幻读发生在当两个完全相同的查询执行时，第二次查询所返回的结果集跟第一个查询不相同。

发生的情况：没有范围锁。

例子：

|  |  |
| --- | --- |
| 事务1 | 事务2 |
| SELECT  \* FROM  users  WHERE  age BETWEEN  10  AND  30 |  |
|  | INSERT  INTO  users VALUES  (  3  , 'Bob'  , 27  );  COMMIT; |
| SELECT  \* FROM  users WHERE  age BETWEEN  10  AND  30; |  |

如何避免：实行序列化隔离模式，在任何一个低级别的隔离中都可能会发生。

**不可重复读**

在基于锁的并行控制方法中，如果在执行select时不添加读锁，就会发生不可重复读问题。

在多版本并行控制机制中，当一个遇到提交冲突的事务需要回退但却被释放时，会发生不可重复读问题。

|  |  |
| --- | --- |
| 事务1 | 事务2 |
| SELECT  \* FROM  users WHERE  id = 1; |  |
|  | UPDATE  users SET  age = 21  WHERE  id = 1  ;  COMMIT; /\* in multiversion concurrency\*/  control, or lock-based READ COMMITTED \* |
| SELECT  \* FROM  users WHERE  id = 1; |  |
|  | COMMIT; /\* lock-based REPEATABLE READ \*/ |

在上面这个例子中，事务2提交成功，它所做的修改已经可见。然而，事务1已经读取了一个其它的值。在序列化和可重复读的隔离级别中，数据库管理系统会返回旧值，即在被事务2修改之前的值。在提交读和未提交读隔离级别下，可能会返回被更新的值，这就是“不可重复读”。

有两个策略可以防止这个问题的发生：

1. 推迟事务2的执行，直至事务1提交或者回退。这种策略在使用锁时应用。(悲观锁机制，比如用select for update为数据行加上一个排他锁)

2. 而在多版本并行控制中，事务2可以被先提交。而事务1，继续执行在旧版本的数据上。当事务1终于尝试提交时，数据库会检验它的结果是否和事务1、事务2顺序执行时一样。如果是，则事务1提交成功。如果不是，事务1会被回退。（乐观锁机制）

**脏读**

脏读发生在一个事务A读取了被另一个事务B修改，但是还未提交的数据。假如B回退，则事务A读取的是无效的数据。这跟不可重复读类似，但是第二个事务不需要执行提交。

|  |  |
| --- | --- |
| 事务1 | 事务2 |
| SELECT  \* FROM  users WHERE  id = 1； |  |
|  | UPDATE  users SET  age = 21  WHERE  id = 1 |
| SELECT  FROM  users WHERE  id = 1； |  |
|  | COMMIT; /\* lock-based DIRTY READ \*/ |