链滴

MySQL 技术内幕 | 事务篇

作者：douniwan

原文链接：https://ld246.com/article/1582220958271

来源网站：链滴

<h3 id="事务的目的">事务的目的</h3>
要么全都执行，要么全都不执行。
<h3 id="事务的分类">事务的分类</h3>
从理论角度，将事务分为：

扁平事务（flat transactions）
带有保存点的扁平事务（flat transactions with savepoints）
链事务（chained transactions）
嵌套事务 （nested transactions）
分布式事务（distributed transaction）

<h4 id="扁平事务">扁平事务</h4>
是事务类型中最简单的一种，但在实际生产环境中，这可能是使用最为频繁的事务。它由 <code>BRGIN WORK</code> 开始，<code>COMMIT WORK</code> 或者 <code>ROLLBACK WOR</code> 结束。处于之间的操作要么都执行要么都不执行。
<h4 id="带有保存点的扁平事务">带有保存点的扁平事务</h4>
指支持扁平事务支持的操作外，允许在事务执行过程中回滚到同一事务中较早的一个状态。通过事务内部使用 <code>SAVEPOINT 节点名字</code> 来建立一个节点，然后可以通过 <code>ROLBACK TO SAVEPOINT 节点名称</code> 来回滚到某节点。

对应扁平事务来说，其隐式地设置了一个保存点（事务开始时的状态
保存点用 SAVE WORK 函数来建立，通知系统记录当前的处理状态
保存点在事务内部是单调递增的，ROLLBACK 不影响保存点的计数

当出现问题时，可以选择回滚到最近一个保存点或者更早的保存点。例如通过 <code>ROLLBAC TO SAVEPOINT 节点名称</code>，回退到节点名称这个事务状态。此时事务不是正常的回滚，还活跃的（事务没有结束，此时执行了节点名称前的所有操作）。如果确认节点前的操作无误，那么就以使用 <code>COMMIT</code> 提交。或者使用 <code>ROLLBACK</code> 回滚掉整个事务
<h4 id="链事务">链事务</h4>
可视为保存点模式的一种变种。

带有保存点的扁平事务中的保存点时易失的，而非持久的，系统崩溃时，索引保存点都将消失

意味着当进行恢复时，事务需要从开始处重新执行，而不能从最近的一个保持点继续执行

链事务的思想

提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式传给下一个开始的事务
提交事务操作和开始下一个事务操作将合并为一个原子操作

与带有保存点的扁平事务不同之处

带有保存点的扁平事务能回滚到任务正确的保存点，而链事务中的回滚仅限于当前事务

即只能恢复到最近的一个的保存点

链事务在执行 COMMIT 后释放了当前事务所持有的锁，而前者不影响迄今为止所持有的锁

<h4 id="嵌套事务">嵌套事务</h4>
是一个层次结构框架，又一个顶层事务控制各个层次的事务（子事务，控制每一个局部的变换</
>

嵌套事务的层次结构

嵌套事务的定义

嵌套事务是由若干事务组成的一棵树，子树既可以是嵌套事务，也可以说是扁平事务
处在叶节点的事务是扁平事务（子事务从根到叶节点的距离可以是不同的
处于根节点的事务称为顶层事务，其他事务称为子事务（事务的前驱称为父事务，下一层称为儿事务
子事务既可以提交也可以回滚（但它的提交操作并不马上生效，除非其父事务已经提交
树中的任意一个事务的回滚会引起它的所有子事务一同回滚（子事务仅保留 A、C、I 特性，不具 D

在 Moss 的理论中，实际的工作是交由叶子节点来完成的，高层的事务仅负责逻辑控制

即只有叶子节点的事务才能访问数据库、发送消息、获取其他类型的资源
高层事务绝对何时调用相关的子任务

<h4 id="分布式事务">分布式事务</h4>
通常是一个在分布式环境下运行的扁平事务，需要根据数据所在位置访问网络中的不同节点

对于分布式事务，其同样需要满足 ACID 特性，要么都发生，要么都失效

<h3 id="事务控制语句">事务控制语句</h3>
在 MySQL 命令行的默认设置下，事务都是自动提交（auto commit）的，即执行完语句后就会上执行 <code>COMMIT</code> 操作,可以通过执行 <code>select @@autocommit；</code> 产看当前的事务是否为自动提交，值为 1 代表是自动提交，值为 0 则代表不是。
可以执行 <code>set @@autocommit=0</code> 或者 <code>set autocommit=0</code> 掉自动提交事务。这时在窗口一执行 <code>insert</code> 操作后，再打开一个窗口二执行 <cod>select</code> 并不会看见窗口一插入的内容，此时等待窗口一 <code>commit</code> 后窗口就能查询到了。如果是自动提交的话，那么只要窗口一插入数据，窗口二就能查询到。
因此需要显式的开启事务需要使用命令 <code>BEGIN</code> 或者 <code>START TRANSATION</code>。输入这两个命令后，再这之后与 <code>ROLLBACK</code> 或 <code>COMMI</code> 之前的操作就属于事务操作了。
<h3 id="相关语句">相关语句</h3>
<code>BEGIN</code>|<code>START TRANSACTION</code>
显式的开启一个事务
<code>COMMIT</code>|<code>COMMIT WORK</code>
都用来提交事务，不同的在于 <code>COMMIT WORK</code> 用来控制事务提交后的行为是 CHAN 还是 RELEASE 的。如果是 CHAIN 方式，那么事务就变成了链事务。可以通过参数 completion_type 来进行控制，默认为 0 表示没有任何操作，这种时候 <ode>COMMIT</code> 和 <code>COMMIT WORK</code> 是完全等价的。为 1 时</trong> <code>COMMIT WORK</code> 等同于 <code>COMMIT AND CHAIN</code>,表示上开启一个相同隔离级别的事务。为 2 时 <code>COMMIT WORK</code> 同于 <code>COMMIT AND RELEASE</code>，当事务提交后自动与服务器断开连接。

<code>ROLLBACK</code>|<code>ROLLBACK WORK</code>
都用来回滚事务。结束正在执行的事务，并撤销所有还没提交的修改。
<code>SAVEPOINT identifier</code>
SAVEPOINE 允许在事务当中创建一个保存点，一个事务中可以创建多个保存点。
<code>RELEASE SAVEPOINT identifier</code>
删除事务的一个保存点，当没有一个该保存点时抛出异常。
<code>ROLLBACK TO [SAVEPOINT] identifier</code>
与 <code>SAVEPOINT identifier</code> 配合使用，用来将事务回滚到某个 identifier 的状态。<
p>
<code>SET TRANSACTION</code>
设置事务的隔离级别，InnoDB 中提供了 READ UNCOMMITTED，READ COMMITTED，REPEATA
LE READ，SERIALIZABLE 四种隔离级别。InnoDB 默认为 REPEATABLE READ。
<h3 id="事务的隔离级别">事务的隔离级别</h3>
事务隔离级别：

读未提交（read　uncommitted）
读提交（read committed）
可重复读（repeatable read）
串行化（serializable ）

读未提交（读取未提交内容）
读未提交，任何操作都不加锁，所以能读到其他事务修改但未提交的数据行，也称之为脏读（Dirty R
ad）。
读提交（读取提交内容，解决了出现脏读的问题)
读操作不加锁，写操作加锁。读被加锁的数据时，读事务每次都读 undo log 中的最近版本，因此可
对同一数据读到不同的版本（不可重复读），但能保证每次都读到最新的数据（事务提交之后的，不
重复读，两次读不一致），但是不会在记录之间加间隙锁，所以允许新的记录插入到被锁定记录的附
，所以再多次使用查询语句时，可能得到不同的结果。
可重复读(可重读，解决不可重复读)
第一次读数据的时候就将数据加行锁（共享锁），使其他事务不能修改当前数据，即可实现可重复读
但是不能锁住 insert 进来的新的数据，当前事务读取或者修改的同时，另一个事务还是可以 insert
交，造成幻读；
（注：mysql 的可重复读的隔离级别解决了 “不可重复读” 和 “幻读” 2 个问题，因为使用了间隙
。）
串行化(可串行化，解决幻读问题)
InnoDB 锁表，读锁和写锁阻塞，强制事务串行执行，解决了幻读的问题；
隔离级别与问题对应表如下：
<table>
<thead>
<tr>
<th align="center">隔离级别</th>
<th align="center">脏读（Dirty Read）</th>
<th align="center">不可重复读（NonRepeatable Read）</th>
<th align="center">幻读（Phantom Read）</th>
</tr>
</thead>
<tbody>
<tr>
<td align="center">未提交读（Read uncommitted）</td>
<td align="center">可能</td>
<td align="center">可能</td>
<td align="center">可能</td>
</tr>
<tr>

<td align="center">已提交读（Read committed）</td>
<td align="center">不可能</td>
<td align="center">可能</td>
<td align="center">可能</td>
</tr>
<tr>
<td align="center">可重复读（Repeatable read</td>
<td align="center">不可能</td>
<td align="center">不可能</td>
<td align="center">可能</td>
</tr>
<tr>
<td align="center">可串行化（Serializable ）</td>
<td align="center">不可能</td>
<td align="center">不可能</td>
<td align="center">不可能</td>
</tr>
</tbody>
</table>

SQL 和 SQL2 标准的默认事务隔离级别是 SERIALIZABLE
InnoDB 存储引擎默认支持的隔离级别是 REPEATABLE READ

但与标准 SQL 不同的是：通过使用 Next-Key Lock 锁的算法来避免幻读的产生
即 InnoDB 在默认的默认隔离级别下已经能完全保证事务的隔离性要求（达到 SQL 标准的 SERIALIZABLE 级别)

在 SERIALIZABLE 的事务隔离级别，InnoDB 会对每个 SELECT 语句后自动加上 LOCK IN SHARE MODE,即共享读锁

因此在此隔离级别下，读占用了锁，对一致性的非锁定读不再予以支持
此隔离级别复核数据库理论上的要求，即事务是 well-formed 的，并且是 two-phrased
SERIALIZABLE 的隔离级别主要用于 InnoDB 存储引擎的分布式事务

在 READ COMMITTED 的事务隔离级别下，除唯一性约束检查及外键约束检查需要 gap lock，他情况都不会使用

<h3 id="事务的锁">事务的锁</h3>

根据读写，分为共享锁 S 和排它锁 X
共享锁：即读加锁，不能写并且可并行读
排它锁：写加锁，其他读写都阻塞

根据加锁范围分为表锁、行锁、间隙锁
表锁：锁 整个表，性能开销最大，其他的读写都要挂起
行锁：锁整个行，以默认隔离级别为例：如果是读，那么会上共享锁，不允许，如果是写，那么改行其他事务无论读写都得阻塞
间隙锁：间隙锁分为两种，一种是不包含记录间隙锁(GAP)，一种是包含记录间锁（Next-Key Lock: Gap Lock+Record Lock），比如对于默认隔离级别的 innoDB 下, 比如表 A 的 id 字段有索引, 并且 id 有 3,8,12,20 这几个值,那么该索引可能被上的包含记录间隙锁区间为:(负无

,3)、[3,8)、[8,12)、[12,20)、[20,正无穷)

1.当事务 T1 锁定了 [8,12),[12,20)这 2 个区间时,当插入 15 时,上面的区间变成:
[8,12)、[12,15)、[15,20)
2.但查询索引含有唯一索引时,Next-Key Lock 降级为 Record Lock,仅锁住索引本身的数据行

3.好,现在表 A 的 id 值变成了: 3,8,12,15,20
4.如果执行下列语句:
select * from A where id>16 for update.
InnoDB 会对(16,正无穷) 加锁,
5.但在 read committed 的事务隔离级别下,因为采用 Record Lock,只会锁定 20 这个值
6.如果在此时另外一个事务 T2,插入了 22 这个值,此时, read committed 隔离级别下就会产生"读"的问题
7.但在 InnoDB 默认存储引擎下的 Next-key Lock 模式下,22 是插入是会被阻塞的,直到事务 T1 交后,释放 X 锁,才能提交 22 这值.这样,InnoDB 就这样解决了幻读的问题
<hr>
本文参考
极客时间·MySQL 实战 45 讲
MYSQL 事务的实现
《MySQL 技术内幕： SQL 编程》· 姜承尧