编程迷思

深入学习MySQL事务：ACID特性的实现原理

事务是MySQL等关系型数据库区别于NoSQL的重要方面，是保证数据一致性的重要手段。**本文将首先介绍MySQL事务相关的基础概念，然后介绍事务的ACID特性，并分析其实现原理。**

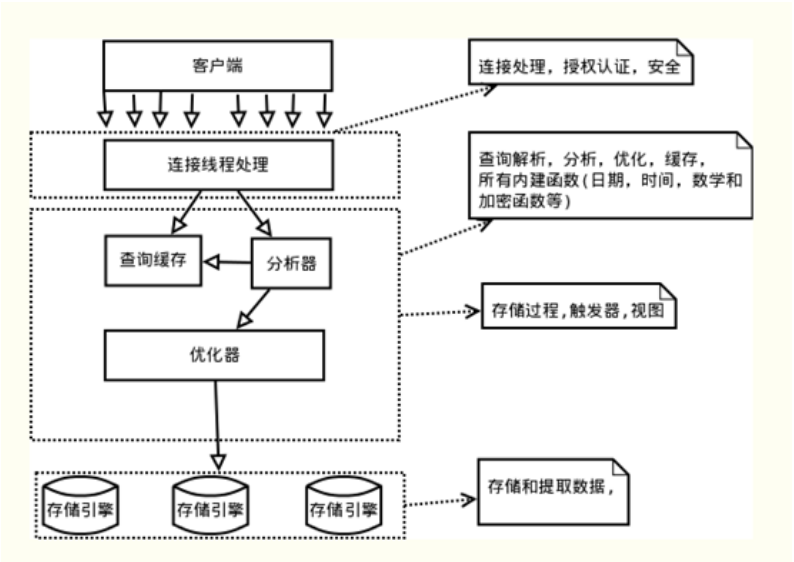
MySQL博大精深，文章疏漏之处在所难免，欢迎批评指正。

一、基础概念

事务（Transaction）是访问和更新数据库的程序执行单元；事务中可能包含一个或多个sql语句，这些语句要么都执行，要么都不执行。作为一个关系型数据库，MySQL支持事务，本文介绍基于MySQL5.6。

首先回顾一下MySQL事务的基础知识。

1. 逻辑架构和存储引擎



图片来源：https://blog.csdn.net/fuzhongmin05/article/details/70904190

如上图所示，MySQL服务器逻辑架构从上往下可以分为三层：

（1）第一层：处理客户端连接、授权认证等。

（2）第二层：服务器层，负责查询语句的解析、优化、缓存以及内置函数的实现、存储过程等。

（3）第三层：存储引擎，负责MySQL中数据的存储和提取。**MySQL中服务器层不管理事务，事务是由存储引擎实现的。**MySQL支持事务的存储引擎有InnoDB、NDB Cluster等，其中InnoDB的使用最为广泛；其他存储引擎不支持事务，如MyIsam、Memory等。

如无特殊说明，后文中描述的内容都是基于InnoDB。

2. 提交和回滚

典型的MySQL事务是如下操作的：

```
start transaction;
……  #一条或多条sql语句
commit;
```

其中start transaction标识事务开始，commit提交事务，将执行结果写入到数据库。如果sql语句执行出现问题，会调用rollback，回滚所有已经执行成功的sql语句。当然，也可以在事务中直接使用rollback语句进行回滚。

自动提交

MySQL中默认采用的是自动提交（autocommit）模式，如下所示：

```
mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
```

在自动提交模式下，如果没有start transaction显式地开始一个事务，那么每个sql语句都会被当做一个事务执行提交操作。

通过如下方式，可以关闭autocommit；需要注意的是，autocommit参数是针对连接的，在一个连接中修改了参数，不会对其他连接产生影响。

```
mysql> set autocommit = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | OFF   |
+---------------+-------+
```

如果关闭了autocommit，则所有的sql语句都在一个事务中，直到执行了commit或rollback，该事务结束，同时开始了另外一个事务。

特殊操作

在MySQL中，存在一些特殊的命令，如果在事务中执行了这些命令，会马上强制执行commit提交事务；如DDL语句(create table/drop table/alter/table)、lock tables语句等等。

不过，常用的select、insert、update和delete命令，都不会强制提交事务。

3. ACID特性

ACID是衡量事务的四个特性：

- 原子性（Atomicity，或称不可分割性）
- 一致性（Consistency）
- 隔离性（Isolation）
- 持久性（Durability）

按照严格的标准，只有同时满足ACID特性才是事务；但是在各大数据库厂商的实现中，真正满足ACID的事务少之又少。例如MySQL的NDB Cluster事务不满足持久性和隔离性；InnoDB默认事务隔离级别是可重复读，不满足隔离性；Oracle默认的事务隔离级别为READ COMMITTED，不满足隔离性……**因此与其说ACID是事务必须满足的条件，不如说它们是衡量事务的四个维度。**

下面将详细介绍ACID特性及其实现原理；为了便于理解，介绍的顺序不是严格按照A-C-I-D。

二、原子性

1. 定义

原子性是指一个事务是一个不可分割的工作单位，其中的操作要么都做，要么都不做；如果事务中一个sql语句执行失败，则已执行的语句也必须回滚，数据库退回到事务前的状态。

2. 实现原理：undo log

在说明原子性原理之前，首先介绍一下MySQL的事务日志。MySQL的日志有很多种，如二进制日志、错误日志、查询日志、慢查询日志等，此外InnoDB存储引擎还提供了两种事务日志：redo log(重做日志)和undo log(回滚日志)。其中redo log用于保证事务持久性；undo log则是事务原子性和隔离性实现的基础。

下面说回undo log。实现原子性的关键，是当事务回滚时能够撤销所有已经成功执行的sql语句。**InnoDB实现回滚，靠的是undo log：当事务对数据库进行修改时，InnoDB会生成对应的undo log；如果事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。**

undo log属于逻辑日志，它记录的是sql执行相关的信息。当发生回滚时，InnoDB会根据undo log的内容做与之前相反的工作：对于每个insert，回滚时会执行delete；对于每个delete，回滚时会执行insert；对于每个update，回滚时会执行一个相反的update，把数据改回去。

以update操作为例：当事务执行update时，其生成的undo log中会包含被修改行的主键(以便知道修改了哪些行)、修改了哪些列、这些列在修改前后的值等信息，回滚时便可以使用这些信息将数据还原到update之前的状态。

三、持久性

1. 定义

持久性是指事务一旦提交，它对数据库的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

2. 实现原理：redo log

redo log和undo log都属于InnoDB的事务日志。下面先聊一下redo log存在的背景。

InnoDB作为MySQL的存储引擎，数据是存放在磁盘中的，但如果每次读写数据都需要磁盘IO，效率会很低。为此，InnoDB提供了缓存(Buffer Pool)，Buffer Pool中包含了磁盘中部分数据页的映射，作为访问数据库的缓冲：当从数据库读取数据时，会首先从Buffer Pool中读取，如果Buffer Pool中没有，则从磁盘读取后放入Buffer Pool；当向数据库写入数据时，会首先写入Buffer Pool，Buffer Pool中修改的数据会定期刷新到磁盘中（这一过程称为刷脏）。

Buffer Pool的使用大大提高了读写数据的效率，但是也带了新的问题：如果MySQL宕机，而此时Buffer Pool中修改的数据还没有刷新到磁盘，就会导致数据的丢失，事务的持久性无法保证。

于是，redo log被引入来解决这个问题：当数据修改时，除了修改Buffer Pool中的数据，还会在redo log记录这次操作；当事务提交时，会调用fsync接口对redo log进行刷盘。如果MySQL宕机，重启时可以读取redo log中的数据，对数据库进行恢复。redo log采用的是WAL（Write-ahead logging，预写式日志），所有修改先写入日志，再更新到Buffer Pool，保证了数据不会因MySQL宕机而丢失，从而满足了持久性要求。

既然redo log也需要在事务提交时将日志写入磁盘，为什么它比直接将Buffer Pool中修改的数据写入磁盘(即刷脏)要快呢？主要有以下两方面的原因：

（1）刷脏是随机IO，因为每次修改的数据位置随机，但写redo log是追加操作，属于顺序IO。

（2）刷脏是以数据页（Page）为单位的，MySQL默认页大小是16KB，一个Page上一个小修改都要整页写入；而redo log中只包含真正需要写入的部分，无效IO大大减少。

3. redo log与binlog

我们知道，在MySQL中还存在binlog(二进制日志)也可以记录写操作并用于数据的恢复，但二者是有着根本的不同的：

（1）作用不同：redo log是用于crash recovery的，保证MySQL宕机也不会影响持久性；binlog是用于point-in-time recovery的，保证服务器可以基于时间点恢复数据，此外binlog还用于主从复制。

13. 【深入学习MySQL】MySQL的索引结构为什么使用B+树？(14)
14. 详解MySQL基准测试和sysbench工具(11)
15. 【Python爬虫】爬了七天七夜，终于爬出了博客园粉丝数排行榜！(7)

推荐排行榜

1. 深入学习Redis（1）：Redis内存模型(252)
2. 深入学习MySQL事务：ACID特性的实现原理(128)
3. 深入学习Redis（2）：持久化(121)
4. 详解Tomcat 配置文件server.xml(114)
5. 深入学习Redis（3）：主从复制(87)

（2）层次不同：redo log是InnoDB存储引擎实现的，而binlog是MySQL的服务器层(可以参考文章前面对MySQL逻辑架构的介绍)实现的，同时支持InnoDB和其他存储引擎。

（3）内容不同：redo log是物理日志，内容基于磁盘的Page；binlog的内容是二进制的，根据binlog_format参数的不同，可能基于sql语句、基于数据本身或者二者的混合。

（4）写入时机不同：binlog在事务提交时写入；redo log的写入时机相对多元：

- 前面曾提到：当事务提交时会调用fsync对redo log进行刷盘；这是默认情况下的策略，修改innodb_flush_log_at_trx_commit参数可以改变该策略，但事务的持久性将无法保证。
- 除了事务提交时，还有其他刷盘时机：如master thread每秒刷盘一次redo log等，这样的好处是不一定要等到commit时刷盘，commit速度大大加快。

四、隔离性

1. 定义

与原子性、持久性侧重于研究事务本身不同，隔离性研究的是不同事务之间的相互影响。隔离性是指，事务内部的操作与其他事务是隔离的，并发执行的各个事务之间不能互相干扰。严格的隔离性，对应了事务隔离级别中的Serializable (可串行化)，但实际应用中出于性能方面的考虑很少会使用可串行化。

隔离性追求的是并发情形下事务之间互不干扰。简单起见，我们主要考虑最简单的读操作和写操作(加锁读等特殊读操作会特殊说明)，那么隔离性的探讨，主要可以分为两个方面：

- (一个事务)写操作对(另一个事务)写操作的影响：锁机制保证隔离性
- (一个事务)写操作对(另一个事务)读操作的影响：MVCC保证隔离性

2. 锁机制

首先来看两个事务的写操作之间的相互影响。隔离性要求同一时刻只能有一个事务对数据进行写操作，InnoDB通过锁机制来保证这一点。

锁机制的基本原理可以概括为：事务在修改数据之前，需要先获得相应的锁；获得锁之后，事务便可以修改数据；该事务操作期间，这部分数据是锁定的，其他事务如果需要修改数据，需要等待当前事务提交或回滚后释放锁。

行锁与表锁

按照粒度，锁可以分为表锁、行锁以及其他位于二者之间的锁。表锁在操作数据时会锁定整张表，并发性能较差；行锁则只锁定需要操作的数据，并发性能好。但是由于加锁本身需要消耗资源(获得锁、检查锁、释放锁等都需要消耗资源)，因此在锁定数据较多情况下使用表锁可以节省大量资源。MySQL中不同的存储引擎支持的锁是不一样的，例如MyIsam只支持表锁，而InnoDB同时支持表锁和行锁，且出于性能考虑，绝大多数情况下使用的都是行锁。

如何查看锁信息

有多种方法可以查看InnoDB中锁的情况，例如：

```
select * from information_schema.innodb_locks; #锁的概况
show engine innodb status; #InnoDB整体状态，其中包括锁的情况
```

下面来看一个例子：

```
#在事务A中执行：
start transaction;
update account SET balance = 1000 where id = 1;
#在事务B中执行：
start transaction;
update account SET balance = 2000 where id = 1;
```

此时查看锁的情况：

```
mysql> select * from information_schema.innodb_locks;
+-------------+-------------+-----------+-----------+-----------------+
| lock_id     | lock_trx_id | lock_mode | lock_type | lock_table      |
+-------------+-------------+-----------+-----------+-----------------+
| 24053:99:3:2 | 24053       | X         | RECORD    | `test`.`account` |
| 24052:99:3:2 | 24052       | X         | RECORD    | `test`.`account` |
+-------------+-------------+-----------+-----------+-----------------+
2 rows in set (0.00 sec)
```

show engine innodb status查看锁相关的部分：

```
---TRANSACTION 24053, ACTIVE 55 sec
1 lock struct(s), heap size 360, 0 row lock(s)
MySQL thread id 108625, OS thread handle 0x7ff8a80e4700, query id 1053844 10.1.90.16 root
---TRANSACTION 24052, ACTIVE 61 sec
2 lock struct(s), heap size 360, 1 row lock(s), undo log entries 1
MySQL thread id 108623, OS thread handle 0x7ff8933f6700, query id 1053833 10.1.90.16 root
```

通过上述命令可以查看事务24052和24053占用锁的情况；其中lock_type为RECORD，代表锁为行锁(记录锁)；lock_mode为X，代表排它锁(写锁)。

除了排它锁(写锁)之外，MySQL中还有共享锁(读锁)的概念。由于本文重点是MySQL事务的实现原理，因此对锁的介绍到此为止，后续会专门写文章分析MySQL中不同锁的区别、使用场景等，欢迎关注。

介绍完写操作之间的相互影响，下面讨论写操作对读操作的影响。

3. 脏读、不可重复读和幻读

首先来看并发情况下，读操作可能存在的三类问题：

（1）脏读：当前事务(A)中可以读到其他事务(B)未提交的数据（脏数据），这种现象是脏读。举例如下（以账户余额表为例）：

时间	事务 A	事务 B
T1	开始事务	开始事务
T2		修改 zhangsan 的余额，将余额由 100 改为 200
T3	查询 zhangsan 的余额，结果为 200【**脏读**】	
T4		提交事务

（2）不可重复读：在事务A中先后两次读取同一个数据，两次读取的结果不一样，这种现象称为不可重复读。脏读与不可重复读的区别在于：前者读到的是其他事务未提交的数据，后者读到的是其他事务已提交的数据。举例如下：

时间	事务 A	事务 B
T1	开始事务	开始事务
T2	查询 zhangsan 的余额，结果为 100	
T3		修改 zhangsan 的余额，将余额由 100 改为 200
T4		提交事务
T5	查询 zhangsan 的余额，结果为 200【**不可重复读**】	

（3）幻读：在事务A中按照某个条件先后两次查询数据库，两次查询结果的条数不同，这种现象称为幻读。不可重复读与幻读的区别可以通俗的理解为：前者是数据变了，后者是数据的行数变了。举例如下：

时间	事务 A	事务 B
T1	开始事务	开始事务
T2	查询 0<id<5 的所有用户的余额： ➢ zhangsan:100 (id=1)	
T3		账户余额表中插入新用户 lisi:200 (id=2)
T4		提交事务
T5	查询 0<id<5 的所有用户的余额： ➢ zhangsan:100 (id=1) ➢ lisi:200 (id=2)**【幻读】**	

4. 事务隔离级别

SQL标准中定义了四种隔离级别，并规定了每种隔离级别下上述几个问题是否存在。一般来说，隔离级别越低，系统开销越低，可支持的并发越高，但隔离性也越差。隔离级别与读问题的关系如下：

隔离级别	脏读	不可重复读	幻读
Read Uncommitted 读未提交	可能	可能	可能
Read Committed 读已提交	不可能	可能	可能
Repeatable Read 可重复读	不可能	不可能	可能
Serializable 可串行化	不可能	不可能	不可能

在实际应用中，**读未提交**在并发时会导致很多问题，而性能相对于其他隔离级别提高却很有限，因此使用较少。**可串行化**强制事务串行，并发效率很低，只有当对数据一致性要求极高且可以接受没有并发时使用，因此使用也较少。因此在大多数数据库系统中，默认的隔离级别是**读已提交(如Oracle)**或**可重复读（后文简称RR）**。

可以通过如下两个命令分别查看全局隔离级别和本次会话的隔离级别：

```
mysql> select @@global.tx_isolation;
+-----------------------+
| @@global.tx_isolation |
+-----------------------+
| REPEATABLE-READ       |
+-----------------------+
1 row in set (0.00 sec)
```

```
mysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set (0.00 sec)
```

InnoDB默认的隔离级别是RR，后文会重点介绍RR。需要注意的是，在SQL标准中，RR是无法避免幻读问题的，但是InnoDB实现的RR避免了幻读问题。

5. MVCC

RR解决脏读、不可重复读、幻读等问题，使用的是MVCC：MVCC全称Multi-Version Concurrency Control，即多版本的并发控制协议。下面的例子很好的体现了MVCC的特点：在同一时刻，不同的事务读取到的数据可能是不同的(即多版本)——在T5时刻，事务A和事务C可以读取到不同版本的数据。

时间	事务 A	事务 B	事务 C
T1	开始事务	开始事务	开始事务
T2	查询 zhangsan 的余额，结果为 100		
T3		修改 zhangsan 的余额，将余额由 100 改为 200	
T4		提交事务	
T5	查询 zhangsan 的余额，结果为 100		查询 zhangsan 的余额，结果为 200

MVCC最大的优点是读不加锁，因此读写不冲突，并发性能好。InnoDB实现MVCC，多个版本的数据可以共存，主要基于以下技术及数据结构：

1）隐藏列：InnoDB中每行数据都有隐藏列，隐藏列中包含了本行数据的事务id、指向undo log的指针等。

2）基于undo log的版本链：前面说到每行数据的隐藏列中包含了指向undo log的指针，而每条undo log也会指向更早版本的undo log，从而形成一条版本链。

3）ReadView：通过隐藏列和版本链，MySQL可以将数据恢复到指定版本；但是具体要恢复到哪个版本，则需要根据ReadView来确定。所谓ReadView，是指事务（记做事务A）在某一时刻给整个事务系统（trx_sys）打快照，之后再进行读操作时，会将读取到的数据中的事务id与trx_sys快照比较，从而判断数据对该ReadView是否可见，即对事务A是否可见。

trx_sys中的主要内容，以及判断可见性的方法如下：

- low_limit_id：表示生成ReadView时系统中应该分配给下一个事务的id。如果数据的事务id大于等于low_limit_id，则对该ReadView不可见。
- up_limit_id：表示生成ReadView时当前系统中活跃的读写事务中最小的事务id。如果数据的事务id小于up_limit_id，则对该ReadView可见。
- rw_trx_ids：表示生成ReadView时当前系统中活跃的读写事务的事务id列表。如果数据的事务id在low_limit_id和up_limit_id之间，则需要判断事务id是否在rw_trx_ids中：如果在，说明生成ReadView时事务仍在活跃中，因此数据对ReadView不可见；如果不在，说明生成ReadView时事务已经提交了，因此数据对ReadView可见。

下面以RR隔离级别为例，结合前文提到的几个问题分别说明。

（1）脏读

时间	事务 A	事务 B
T1	开始事务	开始事务
T2		修改 zhangsan 的余额，将余额由 100 改为 200
T3	查询 zhangsan 的余额，结果为 100 **避免了脏读**	
T4		提交事务

当事务A在T3时刻读取zhangsan的余额前，会生成ReadView，由于此时事务B没有提交仍然活跃，因此其事务id一定在ReadView的rw_trx_ids中，因此根据前面介绍的规则，事务B的修改对ReadView不可见。接下来，事务A根据指针指向的undo log查询上一版本的数据，得到zhangsan的余额为100。这样事务A就避免了脏读。

（2）不可重复读

时间	事务 A	事务 B
T1	开始事务	开始事务
T2	查询 zhangsan 的余额，结果为 100	
T3		修改 zhangsan 的余额，将余额由 100 改为 200
T4		提交事务
T5	查询 zhangsan 的余额，结果为 100 **避免了不可重复读**	

当事务A在T2时刻读取zhangsan的余额前，会生成ReadView。此时事务B分两种情况讨论，一种是如图中所示，事务已经开始但没有提交，此时其事务id在ReadView的rw_trx_ids中；一种是事务B还没有开始，此时其事务id大于等于ReadView的low_limit_id。无论是哪种情况，根据前面介绍的规则，事务B的修改对ReadView都不可见。

当事务A在T5时刻再次读取zhangsan的余额时，会根据T2时刻生成的ReadView对数据的可见性进行判断，从而判断出事务B的修改不可见；因此事务A根据指针指向的undo log查询上一版本的数据，得到zhangsan的余额为100，从而避免了不可重复读。

（3）幻读

时间	事务 A	事务 B
T1	开始事务	开始事务
T2	查询 0<id<5 的所有用户的余额： ➢ zhangsan:100 (id=1)	
T3		账户余额表中插入新用户 lisi:200 (id=2)
T4		提交事务
T5	查询 0<id<5 的所有用户的余额： ➢ zhangsan:100 (id=1) **避免了幻读**	

MVCC避免幻读的机制与避免不可重复读非常类似。

当事务A在T2时刻读取0<id<5的用户余额前，会生成ReadView。此时事务B分两种情况讨论，一种是如图中所示，事务已经开始但没有提交，此时其事务id在ReadView的rw_trx_ids中；一种是事务B还没有开始，此时其事务id大于等于ReadView的low_limit_id。无论是哪种情况，根据前面介绍的规则，事务B的修改对ReadView都不可见。

当事务A在T5时刻再次读取0<id<5的用户余额时，会根据T2时刻生成的ReadView对数据的可见性进行判断，从而判断出事务B的修改不可见。因此对于新插入的数据lisi(id=2)，事务A根据其指针指向的undo log查询上一版本的数据，发现该数据并不存在，从而避免了幻读。

扩展

前面介绍的MVCC，是RR隔离级别下“非加锁读”实现隔离性的方式。下面是一些简单的扩展。

（1）读已提交（RC）隔离级别下的非加锁读

RC与RR一样，都使用了MVCC，其主要区别在于：

RR是在事务开始后第一次执行select前创建ReadView，直到事务提交都不会再创建。根据前面的介绍，RR可以避免脏读、不可重复读和幻读。

RC每次执行select前都会重新建立一个新的ReadView，因此如果事务A第一次select之后，事务B对数据进行了修改并提交，那么事务A第二次select时会重新建立新的ReadView，因此事务B的修改对事务A是可见的。因此RC隔离级别可以避免脏读，但是无法避免不可重复读和幻读。

（2）加锁读与next-key lock

按照是否加锁，MySQL的读可以分为两种：

一种是非加锁读，也称作快照读、一致性读，使用普通的select语句，这种情况下使用MVCC避免了脏读、不可重复读、幻读，保证了隔离性。

另一种是加锁读，查询语句有所不同，如下所示：

```
#共享锁读取
select...lock in share mode
#排它锁读取
select...for update
```

加锁读在查询时会对查询的数据加锁（共享锁或排它锁）。由于锁的特性，当某事务对数据进行加锁读后，其他事务无法对数据进行写操作，因此可以避免脏读和不可重复读。而避免幻读，则需要通过next-key lock。**next-key lock是行锁的一种，实现相当于record lock(记录锁) + gap lock(间隙锁)；其特点是不仅会锁住记录**

本身(record lock的功能)，还会锁定一个范围(gap lock的功能)。因此，加锁读同样可以避免脏读、不可重复读和幻读，保证隔离性。

6. 总结

概括来说，InnoDB实现的RR，通过锁机制（包含next-key lock）、MVCC（包括数据的隐藏列、基于undo log的版本链、ReadView）等，实现了一定程度的隔离性，可以满足大多数场景的需要。

不过需要说明的是，RR虽然避免了幻读问题，但是毕竟不是Serializable，不能保证完全的隔离，下面是两个例子：

第一个例子，如果在事务中第一次读取采用非加锁读，第二次读取采用加锁读，则如果在两次读取之间数据发生了变化，两次读取到的结果不一样，因为加锁读时不会采用MVCC。

第二个例子，如下所示，大家可以自己验证一下。

时间	事务 A	事务 B
T1	开始事务	开始事务
T2	查询 0<id<5 的所有用户的余额： ➢ zhangsan:100 (id=1)	
T3		账户余额表中插入新用户 lisi:200 (id=2)
T4		提交事务
T5	修改 0<id<5 的所有用户的余额为 300	
T6	提交事务 事务A提交后，zhangsan和lisi的余额都变为300	

五、一致性

1. 基本概念

一致性是指事务执行结束后，**数据库的完整性约束没有被破坏，事务执行的前后都是合法的数据状态。**数据库的完整性约束包括但不限于：实体完整性（如行的主键存在且唯一）、列完整性（如字段的类型、大小、长度要符合要求）、外键约束、用户自定义完整性（如转账前后，两个账户余额的和应该不变）。

2. 实现

可以说，一致性是事务追求的最终目标：前面提到的原子性、持久性和隔离性，都是为了保证数据库状态的一致性。此外，除了数据库层面的保障，一致性的实现也需要应用层面进行保障。

实现一致性的措施包括：

- 保证原子性、持久性和隔离性，如果这些特性无法保证，事务的一致性也无法保证
- 数据库本身提供保障，例如不允许向整形列插入字符串值、字符串长度不能超过列的限制等
- 应用层面进行保障，例如如果转账操作只扣除转账者的余额，而没有增加接收者的余额，无论数据库实现的多么完美，也无法保证状态的一致

六、总结

下面总结一下ACID特性及其实现原理：

- 原子性：语句要么全执行，要么全不执行，是事务最核心的特性，事务本身就是以原子性来定义的；实现主要基于undo log
- 持久性：保证事务提交后不会因为宕机等原因导致数据丢失；实现主要基于redo log
- 隔离性：保证事务执行尽可能不受其他事务影响；InnoDB默认的隔离级别是RR，RR的实现主要基于锁机制（包含next-key lock）、MVCC（包括数据的隐藏列、基于undo log的版本链、ReadView）
- 一致性：事务追求的最终目标，一致性的实现既需要数据库层面的保障，也需要应用层面的保障

参考文献

《MySQL技术内幕：InnoDB存储引擎》

《高性能MySQL》

《MySQL运维内参》

https://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_acid

https://dev.mysql.com/doc/refman/5.6/en/innodb-next-key-locking.html

http://blog.sina.com.cn/s/blog_499740cb0100ugs7.html

https://mp.weixin.qq.com/s/2dwGBTmu_da2x-HiHlN0vw

http://www.cnblogs.com/chenpingzhao/p/5065316.html

https://juejin.im/entry/5ba0a254e51d450e735e4a1f

http://hedengcheng.com/?p=771

http://mysql.taobao.org/monthly/2018/03/01/

https://blog.csdn.net/nmjhehe/article/details/98470570

https://elsef.com/2019/03/10/MySQL%E5%A6%82%E4%BD%95%E8%A7%A3%E5%86%B3%E5%90%84%E7%A7%8D%E4%B8%8D%E6%AD%A3%E5%B8%B8%E8%AF%BB/

https://www.zhihu.com/question/372905832

创作不易，如果文章对你有帮助，就点个赞、评个论呗~

创作不易，如果文章对你有帮助，就点个赞、评个论呗~

创作不易，如果文章对你有帮助，就点个赞、评个论呗~

分类： MySQL

标签： MySQL , 事务 , ACID , 原子性 , undo log , 持久性 , redo log , 隔离性 , MVCC , 隔离级别

编程迷思
关注 - 2
粉丝 - 965
+加关注

« 上一篇： 深入学习Redis（5）：集群
» 下一篇： 【BAT面试题系列】面试官：你了解乐观锁和悲观锁吗？

posted @ 2019-01-29 08:26 编程迷思 阅读(45513) 评论(114) 编辑 收藏