# MySQL—事务篇

## 事务隔离级别是怎么实现的?

### 事务有哪些特性?

**事务的四大特性 : ACID**

- **原子性（Atomicity）**: 一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节, 不会出现 a 扣钱了, b 没加钱
- **一致性（Consistency）**： 是指事务操作前和操作后，数据满足完整性约束，数据库保持一致性状态, 总钱数不会少
- **隔离性（Isolation）**: 数据库允许多个**并发事务同时对其数据进行读写和修改**的能力, 每个事务都有一个完整的数据空间，对其他并发事务是隔离的, 也就是 a 购买不会导致 b 购买
- **持久性（Durability）**： 事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失

### InnoDB 引擎通过什么技术来保证事务的这四个特性的呢?

- 原子性 : 通过undo log (回滚日志) 实现
- 隔离性 : 通过 MVCC (多并发版本控制) 或锁机制保证
- 持久性 : 通过 redo log (重做日志) 实现
- 一致性 : 通过原子性 + 隔离性 + 持久性实现

### 并行事务会引发什么问题?

**同时处理多个事务的时候，就可能出现脏读（dirty read）、不可重复读（non-repeatable read）、幻读（phantom read）的问题**

#### 脏读

如果一个事务「读到」了**另一个「未提交事务修改过的数据」**，就意味着发生了「脏读」现象

#### 不可重复读

在一个事务内多次读取同一个数据，如果出现**前后两次读到的数据不一样**的情况，就意味着发生了「不可重复读」现象

#### 幻读

在一个事务内多次查询某个符合查询条件的**「记录数量」**，如果出现前后两次查询到的记录数量不一样的情况，就意味着发生了「幻读」现象

## 事务的隔离级别有哪些?

- **读未提交（read uncommitted）**，指一个事务**还没提交时**，它做的变更就能被其他事务看到, 可能发生**脏读、不可重复读和幻读**现象
- **读提交（read committed）**，指一个事务**提交之后**，它做的变更才能被其他事务看到, 可能发生**不可重复读和幻读**现象
- **可重复读（repeatable read）**，指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，可能发生**幻读**现象, **但是它很大程度上避免幻读现象**, **MySQL InnoDB 引擎的默认隔离级别**
- **串行化（serializable ）**；会对记录加上读写锁，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行

**这四种隔离级别具体是如何实现的呢?**

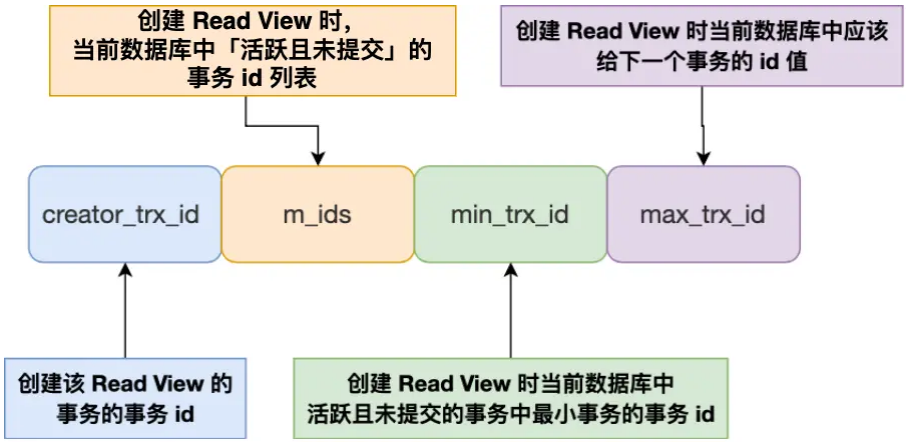
**读未提交**：直接读取最新的数据

**串行化**：读写锁

**读提交和可重复读**：通过 **Read View 来实现**，区别在于生成 Read View 的时机不同, **读提交是在「每个语句执行前」都会重新生成一个 Read View**, **可重复读是「启动事务时」生成一个 Read View**

## Read View 在 MVCC 里如何工作的?

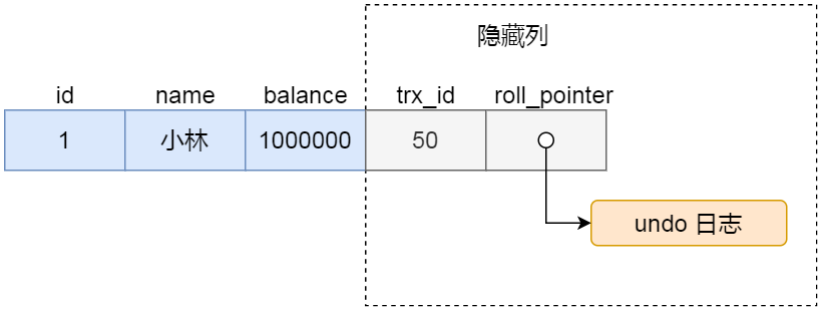
**Read View 四个字段**




- m_ids：指的是在创建 Read View 时，当前数据库中**启动但是未提交的事务 id 列表**，注意是一个列表
- min_trx_id : 指的是在创建 Read View 时, **启动但是未提交的事务 id 列表中事务 id 最小的值**
- max_trx_id：这个并不是 m_ids 的最大值，而是**创建 Read View 时当前数据库中应该给下一个事务的 id 值**，也就是全局事务中最大的事务 id 值 + 1
- creator_trx_id：指的是**创建该 Read View 的事务的事务 id**

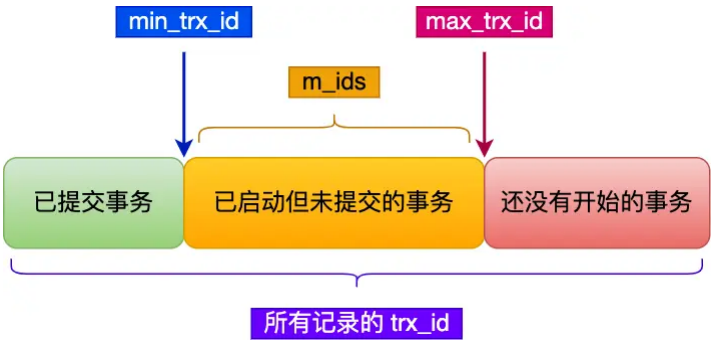
**聚簇索引记录中的两个隐藏列**

知道了 Read View 的字段，我们还需要了解聚簇索引记录中的两个隐藏列




- trx_id : 该事务的事务 id
- roll_pointer : **这个隐藏列是个指针，指向每一个旧版本记录**，可以通过它找到修改前的记录

**MVCC 工作过程**




一个事务去访问记录的时候，除了自己的更新记录总是可见之外，还有这几种情况：

- 如果该记录的 **trx_id 小于 Read View 的 min_trx_id**, 说明该记录是在**创建 Read View 之前的事务**生成的, **可见**
- 如果该记录的 **trx_id 大于 Read View 的 max_trx_id**, 说明该记录是在**创建 Read View 之后的事务**生成的, **不可见**
- 如果记录的 trx_id 值在 **Read View 的 `min_trx_id` 和 `max_trx_id` 之间**, 需要判断 **trx_id 是否在 m_ids 列表中**
    - 如果 **trx_id 在 m_ids 列表中**, 说明创建该记录的事务还未提交, **不可见**
    - 如果 **trx_id 不在 m_ids 列表中**, 说明创建该记录的事务已经提交, **可见**

当前不可见的可以通过 roll_pointer 指针寻找 undo log 版本链找到事务开始时的数据

**这种通过「版本链」来控制并发事务访问同一个记录时的行为就叫 MVCC（多版本并发控制）**

# MySQL 可重复读隔离级别，完全解决幻读了吗?

## 什么是幻读?

两次查询的结果不同, 出现幻像, 主要针对记录数量, 新增或删除语句。而不可重复读针对更新。

## 快照读是如何避免幻读的?

**普通查询通过 Read View 就可以在 undo log 版本链找到事务开始时的数据，所以事务过程中每次查询的数据都是一样的**，即使中途有其他事务插入了新纪录，是查询不出来这条数据的，所以就很好了避免幻读问题

## 当前读是如何避免幻读的?

MySQL 里**除了普通查询是快照读**，其他都是**当前读**，比如 update、insert、delete，这些语句执行前都会查询最新版本的数据，然后再做进一步的操作, 另外，`select ... for update` 这种查询语句是当前读，每次执行的时候都是读取最新的数据。

**Innodb 引擎为了解决「可重复读」隔离级别使用「当前读」而造成的幻读问题，就引出了间隙锁。**

举个具体例子，场景如下：

| 事务A | 事务B |
| --- | --- |
| begin;<br>select name from t_stu where id > 2<br>**for update** | |
| | begin;<br>insert into t_stu values(5,"小飞", 100);<br>阻塞！ |

事务 A 执行了这面这条锁定读语句后，就在对表中的记录加上 id 范围为 (2, +∞] 的 next-key lock **（next-key lock 是间隙锁+记录锁的组合）**

然后，事务 B 在执行插入语句的时候，判断到插入的位置被事务 A 加了 next-key lock，于是事物 B 会生成一个插入意向锁，**同时进入等待状态，直到事务 A 提交了事务。**这就**避免了由于事务 B 插入新记录而导致事务 A 发生幻读的现象。**

## 幻读被完全解决了吗?

**可重复读隔离级别下虽然很大程度上避免了幻读，但是还是没有能完全解决幻读。**

举个例子:

**第一个发生幻读现象的场景**

**表中此时不存在 id = 5 的记录**, 事务 A 更新 id = 5 这条记录，对没错，**事务 A 看不到 id = 5 这条记录，但是他去更新了这条记录**，然后再次查询 id = 5 的记录，**事务 A 就能看到事务 B 插入的记录了**，此时**「幻读」**发生了

因为这种特殊现象的存在，所以我们认为 **MySQL Innodb 中的 MVCC 并不能完全避免幻读现象**

**第二个发生幻读现象的场景**

- T1 时刻：事务 A 先执行「快照读语句」： select * from t_test where id > 100 得到了 3 条记录。
- T2 时刻：事务 B 往插入一个 id= 200 的记录并提交；
- T3 时刻：事务 A 再执行「当前读语句」 select * from t_test where id > 100 for update 就会得到 4 条记录，此时也发生了幻读现象。

**要避免这类特殊场景下发生幻读的现象的话，就是尽量在开启事务之后，马上执行 select ... for update 这类当前读的语句**，因为它会对记录加 next-key lock，从而避免其他事务插入一条新记录。