MySQL 之 MVCC 多版本并发控制

前言

前面介绍了 `MySQL` 中面试常问到的锁，以及详细介绍了三种较为重要的行级别锁（间隙锁、记录锁、临键锁）。今天我们来介绍 `MySQL` 中 `InnoDB` 存储引擎为了在非锁定读下解决幻读问题，使用的 `MVCC` 机制。

我还是想说那句话，对于我们研发来说。`MVCC` 我们工作中通常是接触不到的，理不理解这个技术基本不会影响我们日常工作，但是现在互联网行业，尤其是 `Java` 日渐衰落，面试门槛也随之变高，所以经常会被问到。另外，这个 `MVCC` 其实理解下来并没有那么难，但是说实话如果让我来设计，我确实设计不出来这个思路。

如无特殊说明本篇文章使用的 `MySQL` 环境为 `MySQL 8.0.32 InnoDB 引擎 RR 隔离级别` 。

为什么需要 MVCC

`MVCC` 全名称 `Multi Version Concurrency Control` 翻译过来叫做多版本并发控制，是为了在 **非锁定读的场景下解决幻读问题** 而生的。

前面我们分析了 `MySQL` 内部提供的各种锁机制，使用记录锁、间隙锁、临键锁等行锁可以达到避免幻读的效果。但是加锁是会降低并发性能的，于是为了在提高并发性能，也就是不加锁的情况下还能避免幻读问题，所以 `MySQL` 的开发者想出了 `MVCC` 的技术方案。

`MVCC` 的设计思想是给一条正在被多个事务修改的行数据设定多个版本，每个修改它的事务都会给它生成一条临时版本记录，然后每个需要读取这条行记录的事务，依据规则从多个版本中读取当前事务应该看到的行记录版本。

引用上篇介绍间隙锁的文章使用的表 `cash_repay_apply` 。假如我们有三个事务更新 `id = 1` 的行记录。

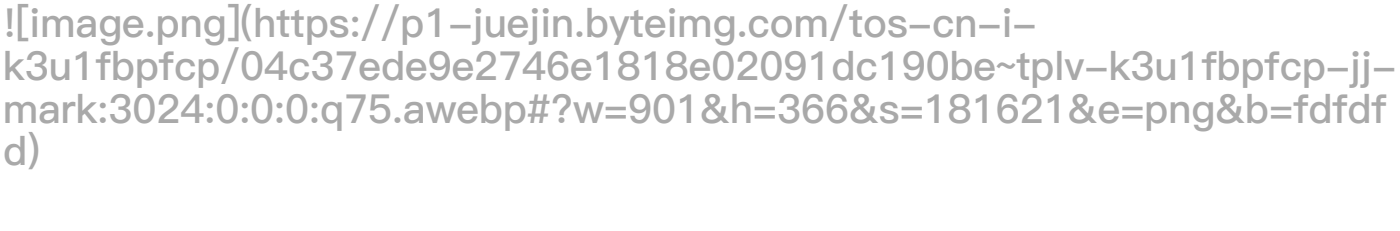
```
-- 初始数据
INSERT INTO `tcbiz_ins`.`cash_repay_apply` (`id`, `member_id`,
`repay_no`) VALUES (1, 1, 'TQYHKN20242231038123');
-- 事务一 假设 trx_id = 6
begin;
update cash_repay_apply set repay_no = '1' where id = 1;

-- 事务二 假设 trx_id = 7
begin;
update cash_repay_apply set repay_no = '2' where id = 1;

-- 事务三 假设 trx_id = 8
begin;
update cash_repay_apply set repay_no = '3' where id = 1;

```

那么这条行数据就有三个 `undo log` 版本（粗略图示，下一段有较为详细图）



锁定读与非锁定读

在介绍 `MVCC` 之前还必须得知道锁定读和非锁定（快照）读，其实在介绍锁的那篇就已经介绍过。这里为了更好的理解 `MVCC` 所以重复介绍一下。

在 `SELECT` 语句后面使用 `FOR UPDATE`、`Lock In Share Mode` 显示加 `X、S 锁` 的方式就叫做锁定读，其他事务的写必须要等待当前 `X、S 锁` 释放。

`SELECT` 不加后缀的普通查询语句都是非锁定（快照）读，即使目标行正在被持有 `X锁`的事务更新也不影响读取，非锁定读会根据规则即时读取目标行的快照，也就是历史版本数据。这里的历史版本就是我们刚刚说的行记录被多个事务修改时产生的不同版本数据。

MVCC 三剑客

`MVCC` 的实现依赖于三个重要角色

* 隐藏字段
* `Undo log` 日志版本链
* `ReadView`

下面我们来依次介绍这三个角色。

行记录隐藏字段 & Undo log 版本链

其实在 `MySQL` 数据表的行记录中除了我们自己定义的字段，还有几个内置隐藏字段。参见官网 [InnoDB Multi-Versioning](http://cxyroad.com/"https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html") 的介绍

官网说了 `InnoDB` 是多版本存储引擎，对数据的修改会保留历史版本，历史版本保留在 `undo log` 中，为了让多个历史版本建立引用关系以及让版本和事务绑定，行记录内部添加了以下字段。

* `DB_ROW_ID` ： 这个字段不是太重要，没有唯一（包括主键）索引的时候才会有用。正常表里都会有主键索引。
* `DB_TRX_ID` ： 更新当前行记录的最新事务 id。
* `DB_ROLL_PTR` ： 回滚指针，指向当前行数据的上一个版本，用它来找到上一个事务更新后产生的历史版本数据

因为 `undo log` 记录的是行数据的多个版本。所以这些字段 `undo log` 日志中也有。

> 前面介绍锁的文章有提到过，`MySQL` 中的 `事务id -> DB_TRX_ID` 是自增的，后开始的 `事务id` 一定大于先开始的 `事务id`。

这时再来完善上面的多事务更新行记录的案例，行记录的版本链如下图


这样我们通过当前行记录的回滚指针 `DB_ROLL_PTR` 就能够遍历到任何一个历史版本。

MVCC 实现原理之 ReadView

`MySQL` 使用 `undo log` 实现了一条行记录的多个版本记录，使用隐藏字段 `DB_TRX_ID、DB_ROLL_PTR` 将这些版本串成链。那么我们需要考虑 **怎样让一个事务看到属于它应该看到的版本呢？** 这肯定需要一个规则。

举个例子，`事务A：trx_id = 7` ，要读取一条行记录 `row`，`row` 有三个版本，其 `DB_TRX_ID` 分别为 `DB_TRX_ID = 6、DB_TRX_ID = 7、DB_TRX_ID = 8`，那么 `事务A` 能读取到 `DB_TRX_ID = 8` 的这个版本吗？显然是不能的，因为 `DB_TRX_ID = 8` 的修改是在事务A 开始之后，那自然不应该看到。那么 `事务A` 能看到 `DB_TRX_ID = 6` 的这个版本吗？不一定，虽然 `DB_TRX_ID = 6` 的版本是在 `事务A` 开始之前产生的，但是还要看 `DB_TRX_ID = 6` 的事务是否已经提交，如果已经提交就可见，如果未提交，则不可见。

`ReadView` 就是用来实现事务可见版本的一个结构体，用这个结构体可以找到对应可见的 `undo log` 版本，当然这借助于一些规则。

实际执行过程，`MySQL` 会先创建一个 `ReadView` 结构体，然后用 `ReadView` 结构体里面的字段值结合规则遍历 `Undo log` 版本链，从最大的 `DB_TRX_ID` 也就是链表头依次寻找，找到第一个符合规则的版本。

`ReadView` 结构中包含了以下字段

* `m_ids` ： `MySQL` 系统当前正在进行中的 `事务id` 集合
* `m_low_limit_id` ： 高水位标识，生成当前 `ReadView` 时，系统即将要分配给下一个新事务的 id 值。当前进行中事务的最大值 + 1（`m_ids 最大元素 + 1`）。`DB_TRX_ID` 大于这个值的 `undo log` 版本都不可见
* `m_up_limit_id` ： 低水位标识，当前进行中事务的最小值。`DB_TRX_ID` 小于这个值的 `undo log` 版本都可以被看见
* `m_creator_trx_id` ： 创建这个 `ReadView` 的事务id，`DB_TRX_ID` 等于这个值的 `undo log` 版本可以被看见

关于这些字段名以及字段解释，网上说法不一，对此我特地下载了 `MySQL 8.0` 版本的源码，我们用源码说话，贴出 `ReadView` 结构体的声明，代码位置 `storage/innobase/include/read0types.h`

```
class ReadView{
    //...
     private:
      /** The read should not see any transaction with trx id >= this
      value. In other words, this is the "high water mark". */
      trx_id_t m_low_limit_id;

      /** The read should see all trx ids which are strictly
      smaller (<) than this value.  In other words, this is the
      low water mark". */
      trx_id_t m_up_limit_id;

      /** trx id of creating transaction, set to TRX_ID_MAX for free
      views. */
      trx_id_t m_creator_trx_id;

      /** Set of RW transactions that was active when this snapshot
      was taken */
      ids_t m_ids;

      /** The view does not need to see the undo logs for transactions
      whose transaction number is strictly smaller (<) than this value:
      they can be removed in purge if not needed by other views */
      trx_id_t m_low_limit_no;

}
```

没时间和你胡闹了，直接搬出官方的权威源码，通过字段的注释能够较好的理解。

> 看完注释之后我更加不理解这个 `m_low_limit_id` 和 `m_up_limit_id` 的命名了，作者难道不觉得这两个名字起的有点怪吗。。。包含 `up` 单词的是低水位标识，包含 `low` 单词的是高水位标识。。。

在事务中，查询语句访问某条记录的时候会先创建 `ReadView`，给 `ReadView` 字段赋值完毕之后，根据字段以及相应规则来搜索可见的 `undo log` 版本 。

ReadView 搜索可见版本的规则

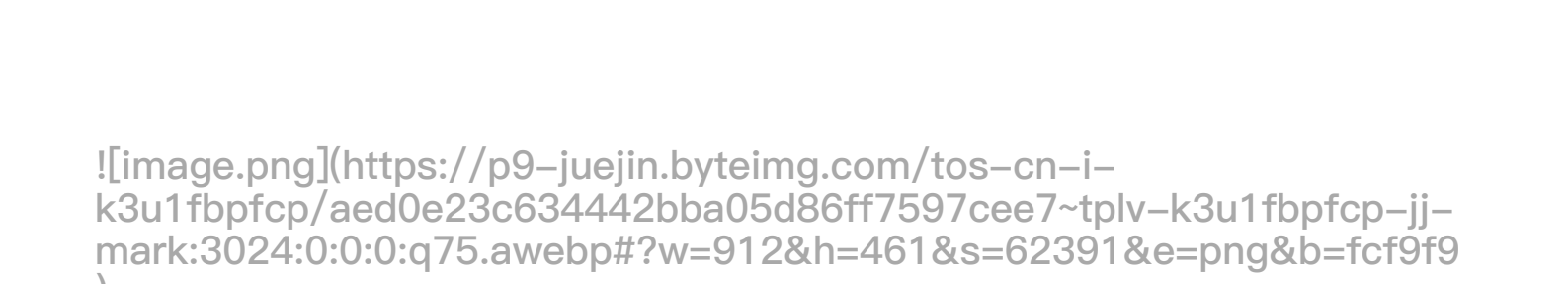
创建完毕之后我们就可以用下面的规则来判断某个版本是否可见，源码中针对索引类型的查询有不同的处理，如果是二级索引还会根据条件回表找聚集索引。这里我们只看聚集索引作为查询条件时 `MVCC` 的 `ReadView` 查询 `Undo log` 版本记录的过程

1. 如果 `undo log` 版本记录的 `DB_TRX_ID` 小于 低水位标识 `m_up_limit_id`。这说明生成这个版本的事务已经提交了，那么这个版本当然是可见的
2. 如果 `undo log` 版本记录的 `DB_TRX_ID` 等于 `m_creator_trx_id`，说明这个版本就是自己申请查询的事务（创建当前 `ReadView` 的事务）所修改产生的，那么这个版本当然是可见的
3. 如果 `undo log` 版本记录的 `DB_TRX_ID` 大于高水位标识 `m_low_limit_id`，说明创建 `ReadView` 的时候这个版本还未生成，那么这个版本当然是不可见的
4. 如果 `undo log` 版本的 `DB_TRX_ID`，位于两者之间。即 `m_up_limit_id ≤ DB_TRX_ID < m_low_limit_id`。那么需要判断 `DB_TRX_ID` 是否存在于 `m_ids` 中，如果存在，说明当前 `ReadView` 创建时，该版本对应的事务还是活跃的，该版本不可见。如果不存在，说明创建 `ReadView` 的过程中，该版本对应的事务已经提交了，该版本可见。

> 注意内存中 m_ids 集合是实时变化的，如果有事务提交或者回滚了，会将 trx_id 从 m_ids 中移除掉。

下面我们通过一个示例来看这个规则，如下图

`事务D trx_id = 9` 执行查询的时候各个属性值是右边文件中标出的那些， 此时 `id = 1` 的行记录的 `Undo log` 版本链如下图



* 根据规则首先找到 `DB_TRX_ID = 8` 的版本，因为 `8` 位于 `m_up_limit_id ≤ DB_TRX_ID < m_low_limit_id` 这条规则，所以查看 `8` 是否还在 `m_ids` 中，发现还在，说明属于未提交事务，该版本不可见。
* 继续找 `DB_TRX_ID = 7` 的版本，因为 `7` 也位于 `m_up_limit_id ≤ DB_TRX_ID < m_low_limit_id` 之间，所以查看发现 `7` 也还在 `m_ids` 中，发现存在，所以该版本也是未提交事务的，也不可见。
* 继续找 `DB_TRX_ID = 6` 的版本，由于 `DB_TRX_ID = 6 < m_up_limit_id`，所以表明创建当前事务的 `ReadView` 时 `DB_TRX_ID = 6` 的事务已经提交了，所以 `DB_TRX_ID = 6` 是可见的版本。所以这里查询到的结果就是 `西瓜`。

上述规则源码位置 `storage/innobase/include/read0types.h` 如下

```
  /** Check whether the changes by id are visible.
  @param[in]    id      transaction id to check against the view
  @param[in]    name    table name
  @return whether the view sees the modifications of id. */
  [[nodiscard]] bool changes_visible(trx_id_t id,const table_name_t
&name) const {
    ut_ad(id > 0);
    if (id < m_up_limit_id || id == m_creator_trx_id) {
      return (true);
    }
    check_trx_id_sanity(id, name);
    if (id >= m_low_limit_id) {
      return (false);
    } else if (m_ids.empty()) {
      return (true);
    }
    const ids_t::value_type *p = m_ids.data();
    return (!std::binary_search(p, p + m_ids.size(), id));
  }
```

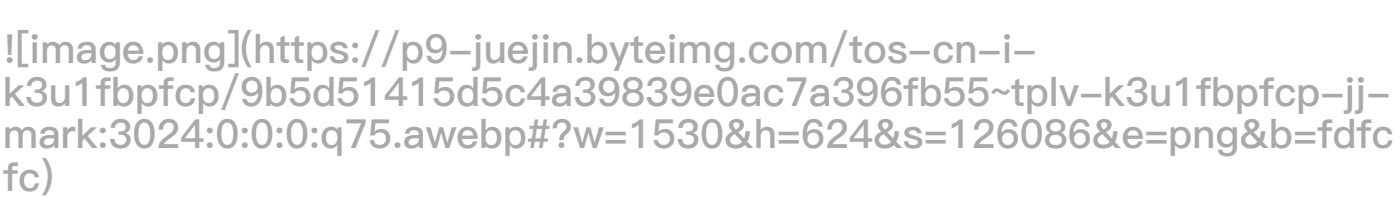
篇幅有限，这里只展示一个示例，其他场景类似，严格按照 `ReadView` 搜索 `undo log` 版本链的规则即可。

MVCC 在 RC 和 RR 下的区别

在 `RC(Read Commited)` 隔离级别下，事务中每次非锁定读的时候都会生成一个 `ReadView`。

在 `RR(Repeatable Read)` 隔离级别下，事务中只有第一次非锁定读的时候会生成一个 `ReadView` ，后续的查询发现当前事务已经存在 `ReadView` 不会重复生成。（特例除外，后面会介绍）

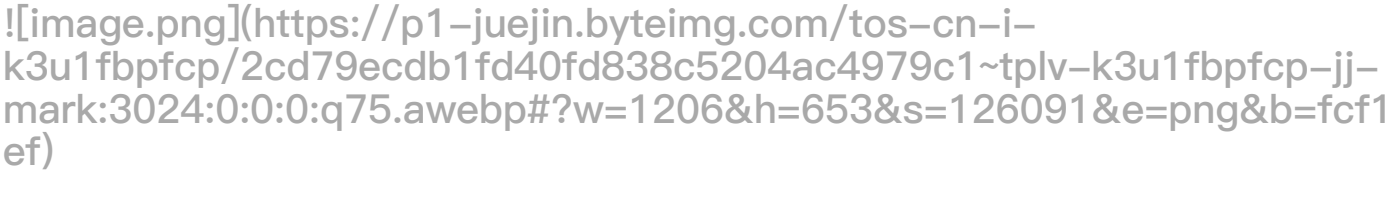
通过下面的示例来了解：



如图，在 `RC` 和 `RR` 的隔离级别下 `事务D` 两次查询得到不同的值。因为 `RC` 的级别下是新的 `ReadView`，`RR` 级别下复用了第一次查询产生的 `ReadView`。

MVCC 真的解决了幻读吗

上面我们介绍 `RC` 和 `RR` 级别下 `ReadView` 的不同时，括号里面标注了特例情况，这个特例情况就是在 `RR` 的隔离级别下，如果两次非锁定读之间夹杂了排他锁操作，那么第二次的 `SELECT` 不会复用第一次 `SELECT` 所产生的 `ReadView`，而是会新建一个 `ReadView`。



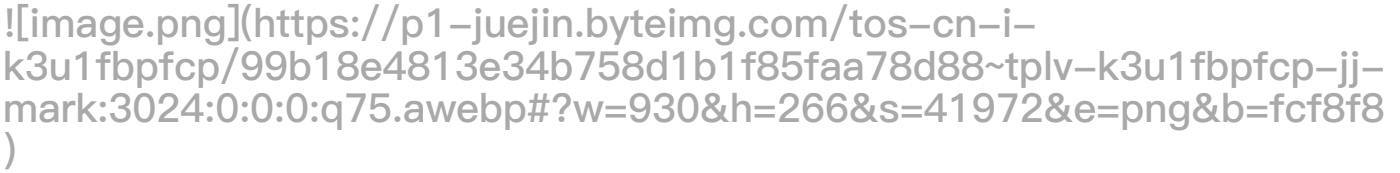
如上图所示，`事务A` 执行第二次 `SELECT` 操作的时候由于 `ReadView` 已经生成了，所以会复用 `ReadView1`，查询结果不变，当执行完黄色的 `update` 语句之后，再次执行 `select * from t_fruit where name = '梨子';`

结果如下

id	name
1	梨子
2	梨子

可以发现当前事务还未提交，却读到了 `事务B` 插入的数据，相当于产生了幻读。这个现象就是因为红色的查询重新建立了 `ReadView`，导致新的 `ReadView` 变成了现在的 `ReadView2`。根据 `ReadView2` 中的字段结合规则，就查询到了这个新的数据。

当执行完黄色部分的 `update` 语句后，`事务B` 插入的那条草莓数据的版本链如下图



> 因为对于插入语句的 `undo log` 在事务提交后就会被删除，所以这里 `undo log` 只有 `事务A` 更新的版本记录。

随后根据红色 `ReadView2` 的规则查找，因为 `当前row` 里面的 `DB_TRX_ID = 6 = m_creator_trx_id` 满足规则，所以这条数据版本对 `ReadView2` 是可见的，所以产生了幻读现象。

所以我们可以发现特定的两次非锁定读之间夹杂排他锁的场景下，`MVCC` 无法解决幻读问题。这是个特例场景。

结语
--

读到这里不用我说大家也能发现，这是一个 `Java` 程序员应该掌握的知识么？我不知道最初是哪个面试官提出这个问题的，我觉得它是个傻鸟，这都需要去读 `C++` 源码才能掌握的知识，让一个 `Java` 程序员掌握，这真的合适么。

。。同时我又觉得这个面试官确实是个大牛，能问出这个问题，说明他自己是掌握的，作为一个 `Java` 程序员他能掌握 `MySQL` 的 `C++` 源码，尊称大牛，在座的各位没有意见吧?

如果这篇文章对你有帮助，记得点赞加！你的支持就是我继续创作的动力！

原文链接: https://juejin.cn/post/7377247135007408162