InnoDB存储引擎对MVCC的实现

Guide 数据库 MySQL 2021年11月6日 约 3750 字

MVCC，全称Multi-Version Concurrency Control，即多版本并发控制。MVCC是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问，在编程语言中实现事务内存。

一致性非锁定读和锁定读

一致性非锁定读

对于 一致性非锁定读（Consistent Nonlocking Reads） 的实现，通常做法是加一个版本号或者时间戳字段，在更新数据的同时版本号 + 1 或者更新时间戳。查询时，将当前可见的版本号与对应记录的版本号进行比对，如果记录的版本小于可见版本，则表示该记录可见

在 `InnoDB` 存储引擎中，多版本控制 (multi versioning) 就是对非锁定读的实现。如果读取的行正在执行 `DELETE` 或 `UPDATE` 操作，这时读取操作不会去等待行上锁的释放。相反地，`InnoDB` 存储引擎会去读取行的一个快照数据，对于这种读取历史数据的方式，我们叫它快照读 (snapshot read)

在 `Repeatable Read` 和 `Read Committed` 两个隔离级别下，如果是执行普通的 `select` 语句（不包括 `select ... lock in share mode`，`select ... for update`）则会使用 `一致性非锁定读（MVCC）`。并且在 `Repeatable Read` 下 `MVCC` 实现了可重复读和防止部分幻读

锁定读

如果执行的是下列语句，就是 锁定读（Locking Reads）

- `select ... lock in share mode`
- `select ... for update`
- `insert`、`update`、`delete` 操作

在锁定读下，读取的是数据的最新版本，这种读也被称为 `当前读（current read）`。锁定读会对读取到的记录加锁：

- `select ... lock in share mode`：对记录加 `S` 锁，其它事务也可以加 `S` 锁，如果加 `x` 锁则会被阻塞
- `select ... for update`、`insert`、`update`、`delete`：对记录加 `X` 锁，且其它事务不能加任何锁

在一致性非锁定读下，即使读取的记录已被其它事务加上 `X` 锁，这时记录也是可以被读取的，即读取的快照数据。上面说了，在 `Repeatable Read` 下 `MVCC` 防止了部分幻读，这边的“部分”是指在 `一致性非锁定读` 情况下，只能读取到第一次查询之前所插入的数据（根据 Read View 判断数据可见性，Read View 在第一次查询时生成）。但是！如果是 `当前读`，每次读取的都是最新数据，这时如果两次查询中间有其它事务插入数据，就会产生幻读。所以，**`InnoDB` 在实现 `Repeatable Read` 时，如果执行的是当前读，则会对读取的记录使用 `Next-key Lock`，来防止其它事务在间隙间插入数据**

InnoDB 对 MVCC 的实现

`MVCC` 的实现依赖于：**隐藏字段、Read View、undo log**。在内部实现中，`InnoDB` 通过数据行的 `DB_TRX_ID` 和 `Read View` 来判断数据的可见性，如不可见，则通过数据行的 `DB_ROLL_PTR` 找到 `undo log` 中的历史版本。每个事务读到的数据版本可能是不一样的，在同一个事务中，用户只能看到该事务创建 `Read View` 之前已经提交的修改和该事务本身做的修改

隐藏字段

在内部，`InnoDB` 存储引擎为每行数据添加了三个 隐藏字段：

- `DB_TRX_ID（6字节）`：表示最后一次插入或更新该行的事务 id。此外，`delete` 操作在内部被视为更新，只不过会在记录头 `Record header` 中的 `deleted_flag` 字段将其标记为已删除
- `DB_ROLL_PTR（7字节）` 回滚指针，指向该行的 `undo log`。如果该行未被更新，则为空
- `DB_ROW_ID（6字节）`：如果没有设置主键且该表没有唯一非空索引时，`InnoDB` 会使用该 id 来生成聚簇索引

ReadView

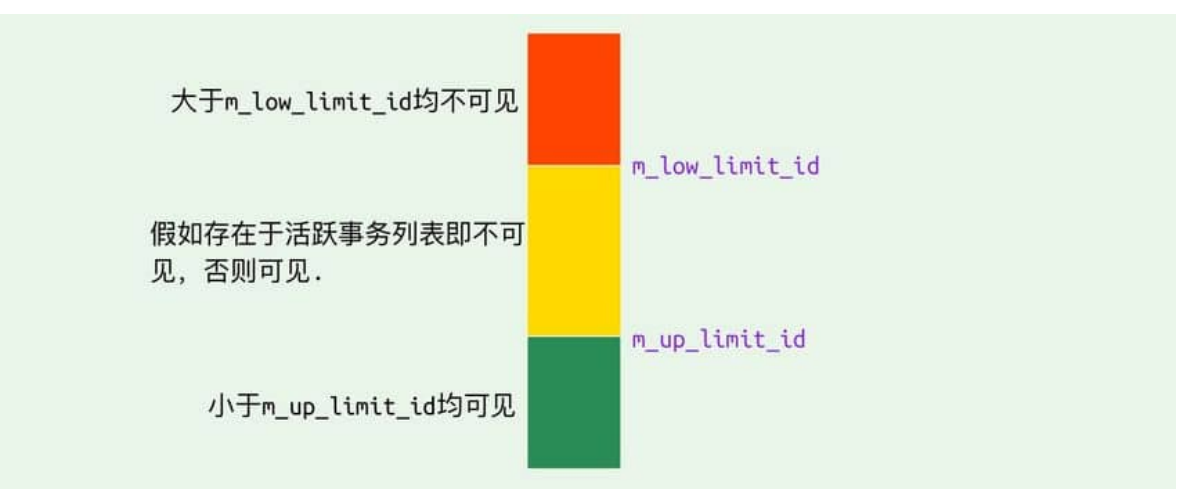
```
class ReadView {
  /* ... */
private:
  trx_id_t m_low_limit_id;      /* 大于等于这个 ID 的事务均不可见 */

  trx_id_t m_up_limit_id;       /* 小于这个 ID 的事务均可见 */

  trx_id_t m_creator_trx_id;    /* 创建该 Read View 的事务ID */

  trx_id_t m_low_limit_no;      /* 事务 Number, 小于该 Number 的 Undo Logs 均可以被 Purge */

  ids_t m_ids;                  /* 创建 Read View 时的活跃事务列表 */

  m_closed;                     /* 标记 Read View 是否 close */
}
```

`Read View` 主要是用来做可见性判断，里面保存了“当前对本事务不可见的其他活跃事务”

主要有以下字段：

- `m_low_limit_id`：目前出现过的最大的事务 ID+1，即下一个将被分配的事务 ID。大于等于这个 ID 的数据版本均不可见
- `m_up_limit_id`：活跃事务列表 `m_ids` 中最小的事务 ID，如果 `m_ids` 为空，则 `m_up_limit_id` 为 `m_low_limit_id`。小于这个 ID 的数据版本均可见
- `m_ids`：`Read View` 创建时其他未提交的活跃事务 ID 列表。创建 `Read View` 时，将当前未提交事务 ID 记录下来，后续即使它们修改了记录行的值，对于当前事务也是不可见的。`m_ids` 不包括当前事务自己和已提交的事务（正在内存中）
- `m_creator_trx_id`：创建该 `Read View` 的事务 ID

事务可见性示意图（图源）：



undo-log

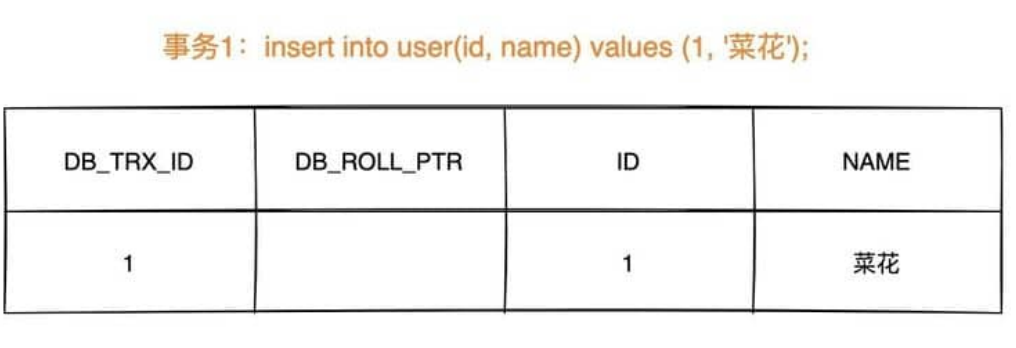
`undo log` 主要有两个作用：

- 当事务回滚时用于将数据恢复到修改前的样子
- 另一个作用是 `MVCC`，当读取记录时，若该记录被其他事务占用或当前版本对该事务不可见，则可以通过 `undo log` 读取之前的版本数据，以此实现非锁定读

在 `InnoDB` 存储引擎中 `undo log` 分为两种：`insert undo log` 和 `update undo log`：

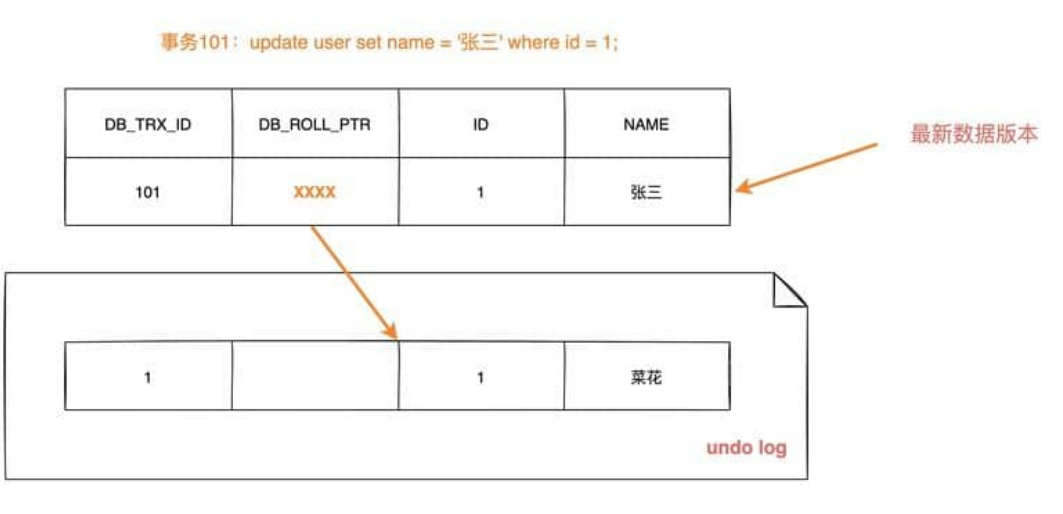
1. **`insert undo log`**：指在 `insert` 操作中产生的 `undo log`。因为 `insert` 操作的记录只对事务本身可见，对其他事务不可见，故该 `undo log` 可以在事务提交后直接删除。不需要进行 `purge` 操作

`insert` 时的数据初始状态：

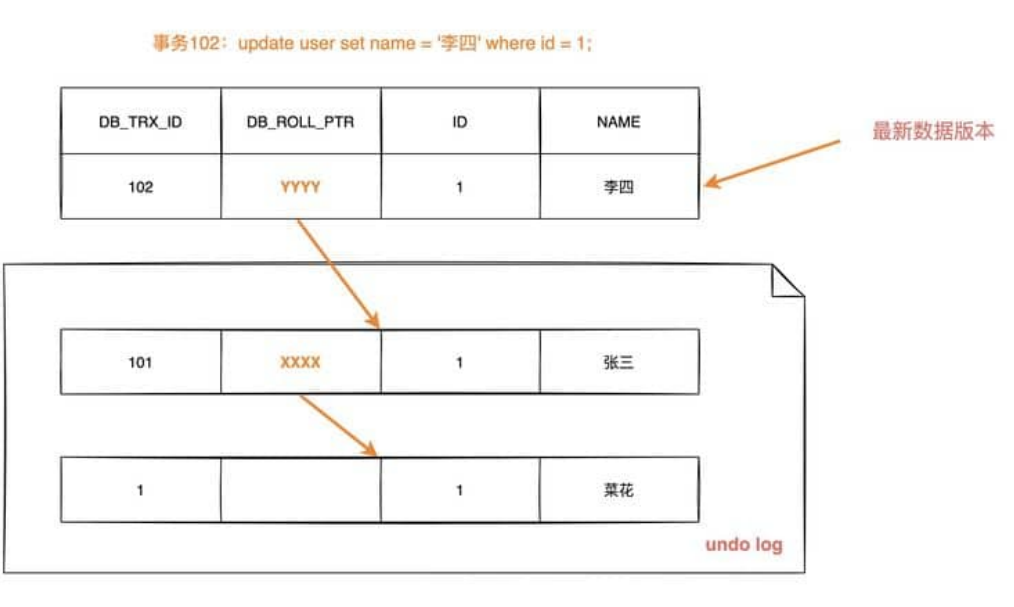


2. **`update undo log`**：`update` 或 `delete` 操作中产生的 `undo log`。该 `undo log` 可能需要提供 `MVCC` 机制，因此不能在事务提交时就进行删除。提交时放入 `undo log` 链表，等待 `purge线程` 进行最后的删除

数据第一次被修改时：



数据第二次被修改时：



不同事务或者相同事务的对同一记录行的修改，会使该记录行的 `undo log` 成为一条链表，链首就是最新的记录，链尾就是最早的旧记录。

数据可见性算法

在 `InnoDB` 存储引擎中，创建一个新事务后，执行每个 `select` 语句前，都会创建一个快照（Read View），**快照中保存了当前数据库系统中正处于活跃（没有 commit）的事务的 ID 号**。其实简单的说保存的是系统中当前不应该被本事务看到的其他事务 ID 列表（即 m_ids）。当用户在这个事务中要读取某个记录行的时候，`InnoDB` 会将该记录行的 `DB_TRX_ID` 与 `Read View` 中的一些变量及当前事务 ID 进行比较，判断是否满足可见性条件

具体的比较算法如下(图源)：

changes_visible() 的返回结果 true 代表可见，false 代表不可见.

```
/* storage/innobase/include/read0types.h */

bool changes_visible(trx_id_t id, const table_name_t &name) const
    MY_ATTRIBUTE((warn_unused_result)) {
  ut_ad(id > 0);

  /* 假如 trx_id 小于 Read View 限制的最小活跃事务ID m_up_limit_id 或者等于正在创建的事务ID
   * m_creator_trx_id 即满足事务的可见性. */
  if (id < m_up_limit_id || id == m_creator_trx_id) {
    return (true);
  }

  /* 检查 trx_id 是否有效. */
  check_trx_id_sanity(id, name);

  if (id >= m_low_limit_id) {
    /* 假如 trx_id 大于最大活跃的事务ID m_low_limit_id, 即不可见. */
    return (false);
  } else if (m_ids.empty()) {
    /* 假如目前不存在活跃的事务, 即可见. */
    return (true);
  }

  const ids_t::value_type *p = m_ids.data();

  /* 利用二分查找搜索活跃事务列表, 当 trx_id 在 m_up_limit_id 和 m_low_limit_id 之间
   * 如果 id 在 m_ids 数组中, 表明 ReadView 创建时候, 事务处于活跃状态, 因此记录不可见. */
  return (!std::binary_search(p, p + m_ids.size(), id));
}
```

1. 如果记录 DB_TRX_ID < m_up_limit_id，那么表明最新修改该行的事务（DB_TRX_ID）在当前事务创建快照之前就提交了，所以该记录行的值对当前事务是可见的
2. 如果 DB_TRX_ID >= m_low_limit_id，那么表明最新修改该行的事务（DB_TRX_ID）在当前事务创建快照之后才修改该行，所以该记录行的值对当前事务不可见。跳到步骤 5
3. m_ids 为空，则表明在当前事务创建快照之前，修改该行的事务就已经提交了，所以该记录行的值对当前事务是可见的
4. 如果 m_up_limit_id <= DB_TRX_ID < m_low_limit_id，表明最新修改该行的事务（DB_TRX_ID）在当前事务创建快照的时候可能处于“活动状态”或者“已提交状态”；所以就要对活跃事务列表 m_ids 进行查找（源码中是用的二分查找，因为是有序的）
   - 如果在活跃事务列表 m_ids 中能找到 DB_TRX_ID，表明：① 在当前事务创建快照前，该记录行的值被事务 ID 为 DB_TRX_ID 的事务修改了，但没有提交；或者 ② 在当前事务创建快照后，该记录行的值被事务 ID 为 DB_TRX_ID 的事务修改了。这些情况下，这个记录行的值对当前事务都是不可见的。跳到步骤 5
   - 在活跃事务列表中找不到，则表明“id 为 trx_id 的事务”在修改“该记录行的值”后，在“当前事务”创建快照前就已经提交了，所以记录行对当前事务可见
5. 在该记录行的 DB_ROLL_PTR 指针所指向的 `undo log` 取出快照记录，用快照记录的 DB_TRX_ID 跳到步骤 1 重新开始判断，直到找到满足的快照版本或返回空

RC 和 RR 隔离级别下 MVCC 的差异

在事务隔离级别 `RC` 和 `RR` （InnoDB 存储引擎的默认事务隔离级别）下，`InnoDB` 存储引擎使用 `MVCC`（非锁定一致性读），但它们生成 `Read View` 的时机却不同

- 在 RC 隔离级别下的 **`每次select`** 查询前都生成一个`Read View` (m_ids 列表)
- 在 RR 隔离级别下只在事务开始后 **`第一次select`** 数据前生成一个`Read View`（m_ids 列表）

MVCC 解决不可重复读问题

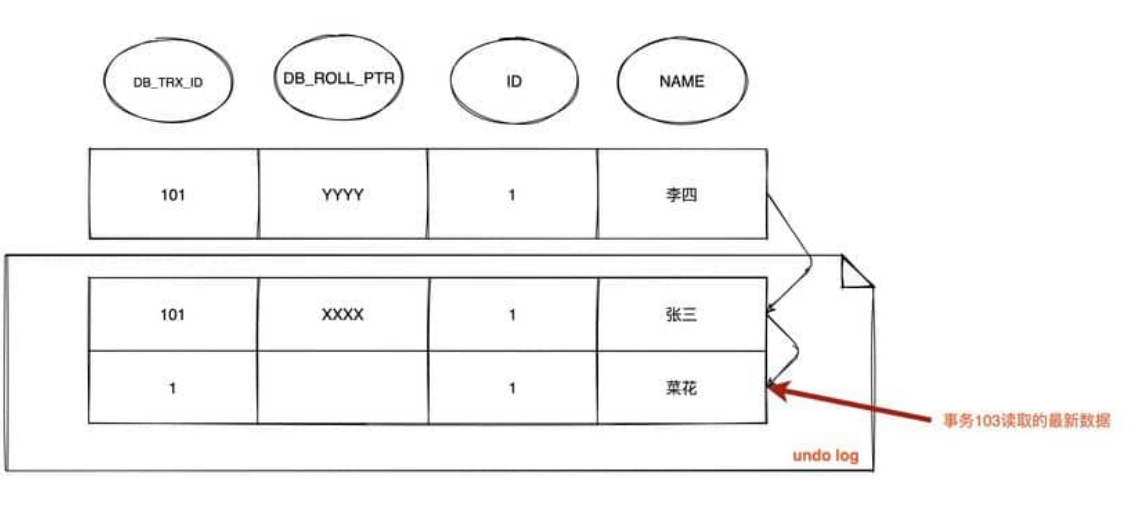
虽然 RC 和 RR 都通过 `MVCC` 来读取快照数据，但由于 **生成 Read View 时机不同**，从而在 RR 级别下实现可重复读

举个例子：

	事务101	事务102	事务103
T1	begin;		
T2		begin;	begin;
T3	update user set name = '张三' where id = 1;		
T4	update user set name = '李四' where id = 1;	...	select * from user where id = 1;
T5	commit;	update user set name = '王五' where id = 1;	
T6			select * from user where id = 1;
T7		update user set name = '赵六' where id = 1;	
T8		commit;	
T9			select * from user where id = 1;
T10			commit;

在 RC 下 ReadView 生成情况

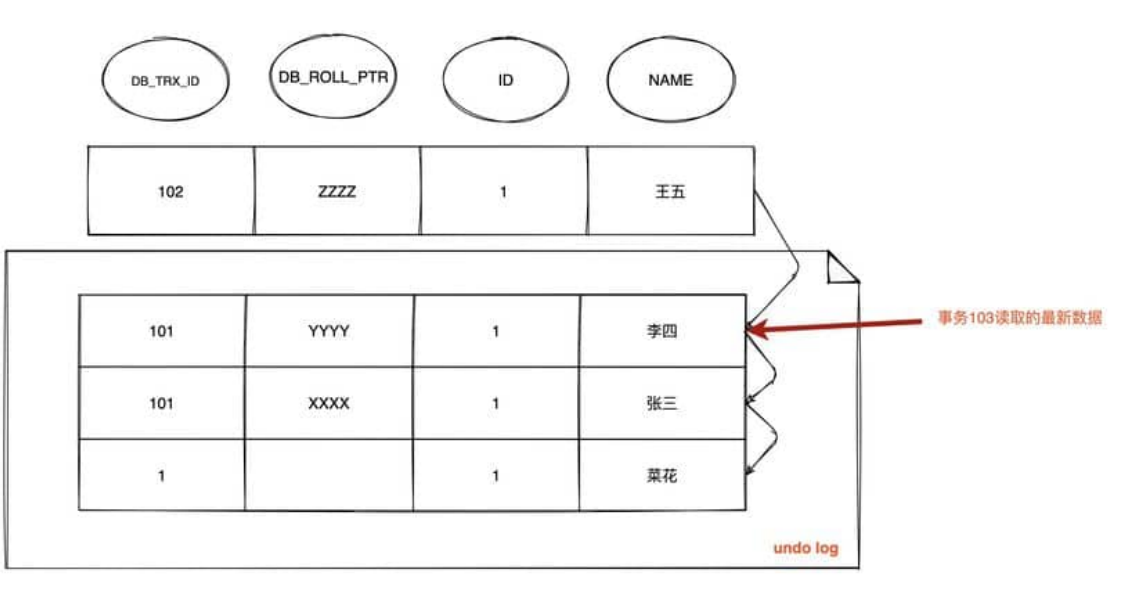
1. 假设时间线来到 T4，那么此时数据行 id = 1 的版本链为：



由于 RC 级别下每次查询都会生成`Read View`，并且事务 101、102 并未提交，此时 `103` 事务生成的 `Read View` 中活跃的事务 **`m_ids` 为：[101,102]**，`m_low_limit_id` 为：104，`m_up_limit_id` 为：101，`m_creator_trx_id` 为：103

- 此时最新记录的 `DB_TRX_ID` 为 101，m_up_limit_id <= 101 < m_low_limit_id，所以要在 `m_ids` 列表中查找，发现 `DB_TRX_ID` 存在列表中，那么这个记录不可见
- 根据 `DB_ROLL_PTR` 找到 `undo log` 中的上一版本记录，上一条记录的 `DB_TRX_ID` 还是 101，不可见
- 继续找上一条 `DB_TRX_ID` 为 1，满足 1 < m_up_limit_id，可见，所以事务 103 查询到数据为 `name = 菜花`

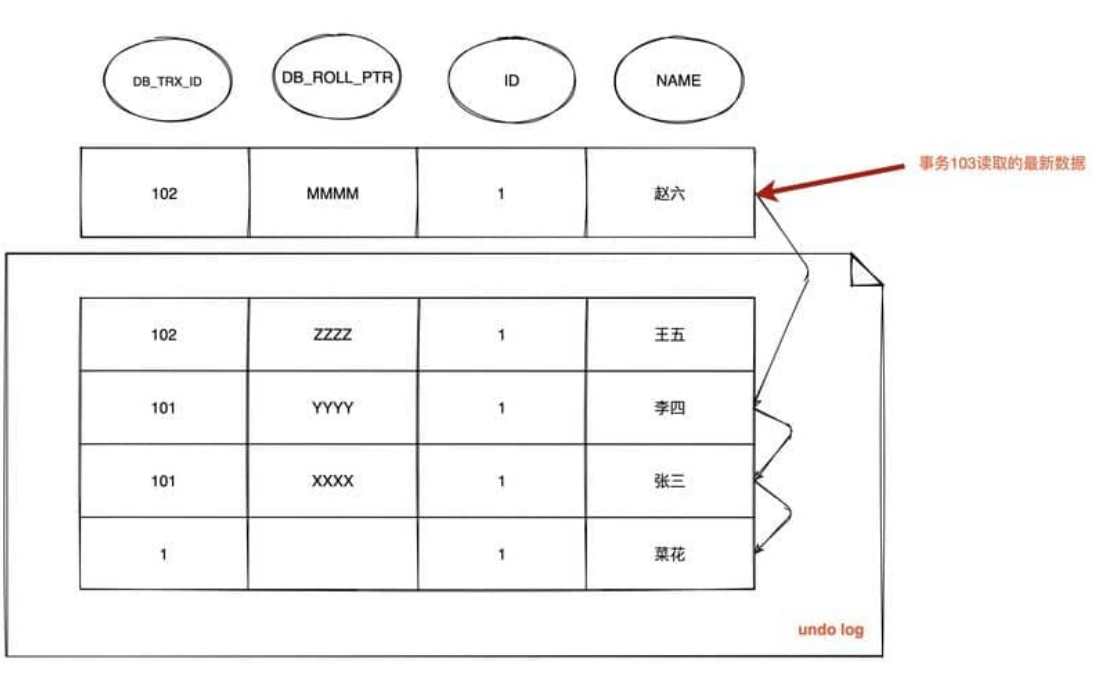
2. 时间线来到 T6，数据的版本链为：



因为在 RC 级别下，重新生成 `Read View`，这时事务 101 已经提交，102 并未提交，所以此时 `Read View` 中活跃的事务 **`m_ids`：[102]**，`m_low_limit_id` 为：104，`m_up_limit_id` 为：102，`m_creator_trx_id` 为：103

- 此时最新记录的 `DB_TRX_ID` 为 102，m_up_limit_id <= 102 < m_low_limit_id，所以要在 `m_ids` 列表中查找，发现 `DB_TRX_ID` 存在列表中，那么这个记录不可见
- 根据 `DB_ROLL_PTR` 找到 `undo log` 中的上一版本记录，上一条记录的 `DB_TRX_ID` 为 101，满足 101 < m_up_limit_id，记录可见，所以在 `T6` 时间点查询到数据为 `name = 李四`，与时间 T4 查询到的结果不一致，不可重复读！

3. 时间线来到 T9，数据的版本链为：



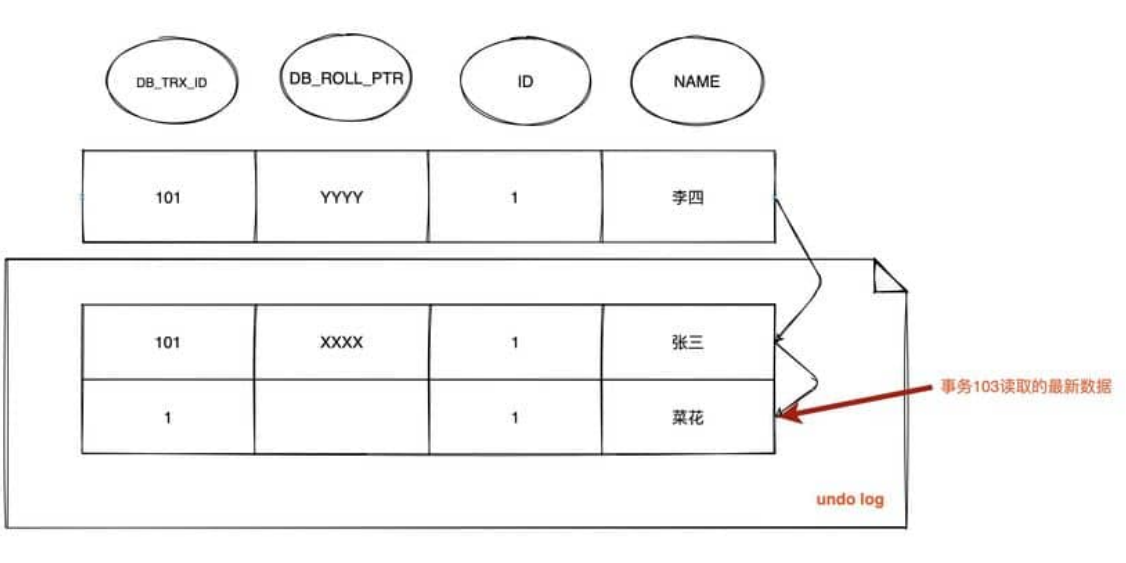
重新生成 `Read View`， 这时事务 101 和 102 都已经提交，所以 **m_ids** 为空，则 m_up_limit_id = m_low_limit_id = 104，最新版本事务 ID 为 102，满足 102 < m_low_limit_id，可见，查询结果为 `name = 赵六`

> **总结：在 RC 隔离级别下，事务在每次查询开始时都会生成并设置新的 Read View，所以导致不可重复读**

在 RR 下 ReadView 生成情况

在可重复读级别下，只会在事务开始后第一次读取数据时生成一个 Read View（m_ids 列表）

1. 在 T4 情况下的版本链为：

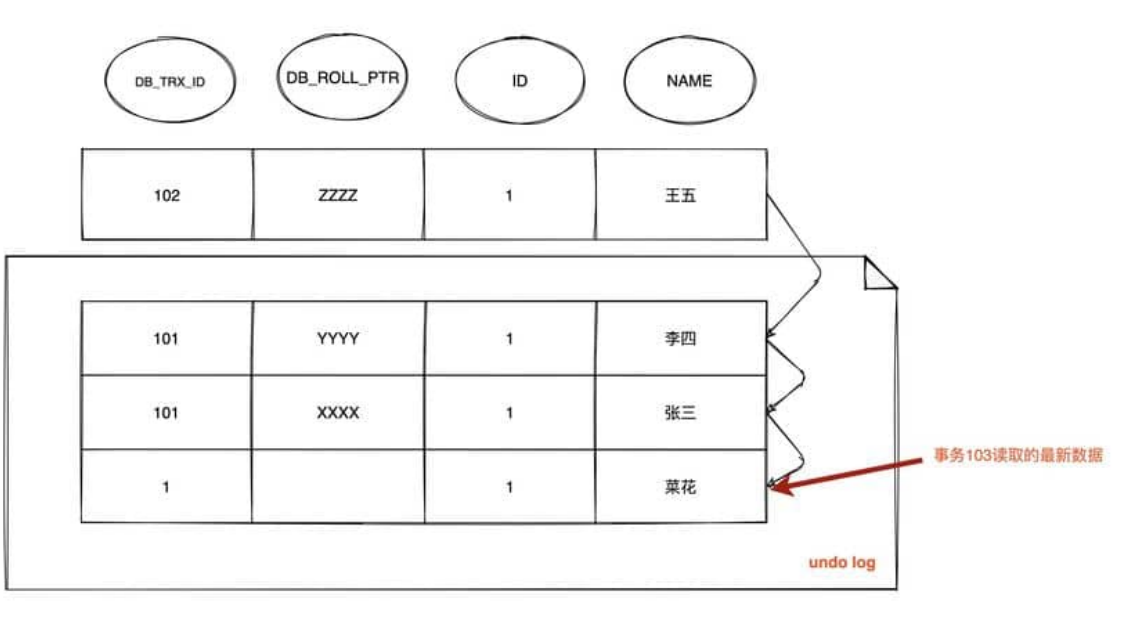


在当前执行 `select` 语句时生成一个 `Read View`，此时 **`m_ids`：[101,102]**，`m_low_limit_id` 为：104，`m_up_limit_id` 为：101，`m_creator_trx_id` 为：103

此时和 RC 级别下一样：

- 最新记录的 `DB_TRX_ID` 为 101，m_up_limit_id <= 101 < m_low_limit_id，所以要在 `m_ids` 列表中查找，发现 `DB_TRX_ID` 存在列表中，那么这个记录不可见
- 根据 `DB_ROLL_PTR` 找到 `undo log` 中的上一版本记录，上一条记录的 `DB_TRX_ID` 还是 101，不可见
- 继续找上一条 `DB_TRX_ID` 为 1，满足 1 < m_up_limit_id，可见，所以事务 103 查询到数据为 `name = 菜花`

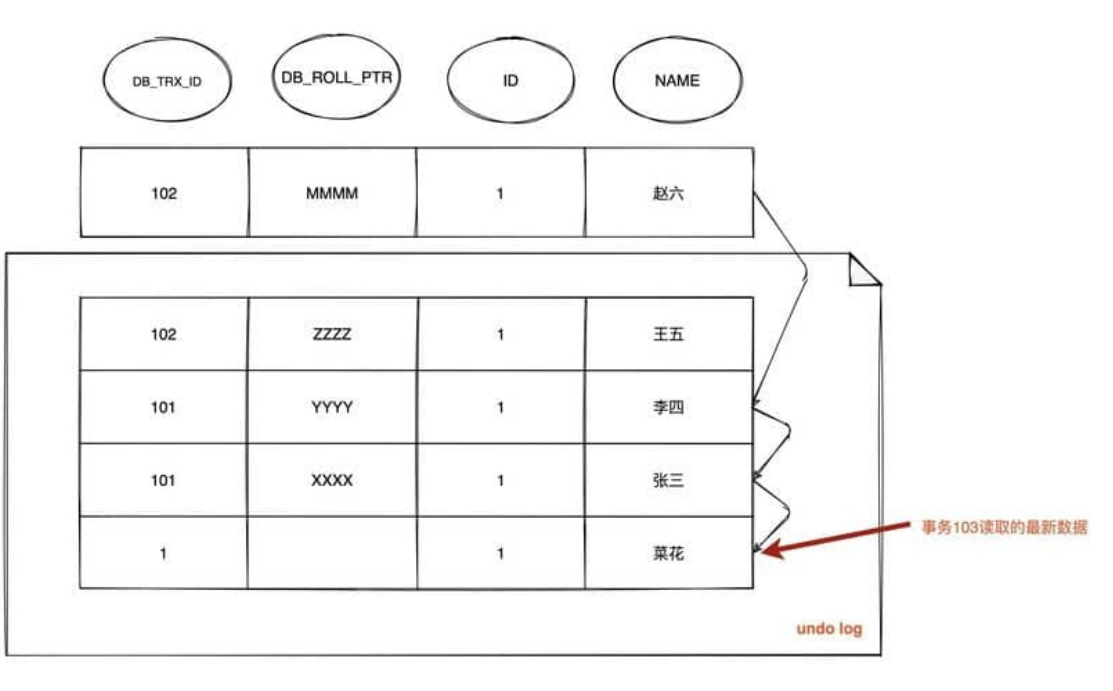
2. 时间点 T6 情况下：



在 RR 级别下只会生成一次`Read View`，所以此时依然沿用 **`m_ids`：[101,102]**，`m_low_limit_id` 为：104，`m_up_limit_id` 为：101，`m_creator_trx_id` 为：103

- 最新记录的 `DB_TRX_ID` 为 102，m_up_limit_id <= 102 < m_low_limit_id，所以要在 `m_ids` 列表中查找，发现 `DB_TRX_ID` 存在列表中，那么这个记录不可见
- 根据 `DB_ROLL_PTR` 找到 `undo log` 中的上一版本记录，上一条记录的 `DB_TRX_ID` 为 101，不可见
- 继续根据 `DB_ROLL_PTR` 找到 `undo log` 中的上一版本记录，上一条记录的 `DB_TRX_ID` 还是 101，不可见
- 继续找上一条 `DB_TRX_ID` 为 1，满足 1 < m_up_limit_id，可见，所以事务 103 查询到数据为 `name = 菜花`

3. 时间点 T9 情况下：



此时情况跟 T6 完全一样，由于已经生成了 `Read View`，此时依然沿用 **`m_ids`：[101,102]**，所以查询结果依然是 `name = 菜花`

MVCC➕Next-key-Lock 防止幻读

`InnoDB` 存储引擎在 RR 级别下通过 `MVCC` 和 `Next-key Lock` 来解决幻读问题：

1、执行普通 `select`，此时会以 `MVCC` 快照读的方式读取数据

在快照读的情况下，RR 隔离级别只会在事务开启后的第一次查询生成 `Read View`，并使用至事务提交。所以在生成 `Read View` 之后其它事务所做的更新、插入记录版本对当前事务并不可见，实现了可重复读和防止快照读下的“幻读”

2、执行 select...for update/lock in share mode、insert、update、delete 等当前读

在当前读下，读取的都是最新的数据，如果其它事务有插入新的记录，并且刚好在当前事务查询范围内，就会产生幻读！`InnoDB` 使用 Next-key Lock 来防止这种情况。当执行当前读时，会锁定读取到的记录的同时，锁定它们的间隙，防止其它事务在查询范围内插入数据。只要我不让你插入，就不会发生幻读

参考

- 《MySQL 技术内幕 InnoDB 存储引擎第 2 版》
- Innodb 中的事务隔离级别和锁的关系
- MySQL 事务与 MVCC 如何实现的隔离级别
- InnoDB 事务分析-MVCC

当前读与快照读

维特无忧堡 关注
1 2018.07.29 17:29:17 字数 814 阅读 27,071

前言

> 在这里记录一下一个博客，觉得写的很好 http://hedengcheng.com/?p=771 ，

概念

快照读

读取的是记录数据的可见版本（可能是过期的数据），不用加锁

当前读

读取的是记录数据的最新版本，并且当前读返回的记录都会加上锁，保证其他事务不会再并发的修改这条记录

概念说的比较虚，也不好理解，接着举一个例子吧，假设你开启了两个事务，分别是A和B，这里有个张表，user表，里面有四条数据

Id	username	password	isAdmin	name	type
1	vision	123456	1	维特无忧堡	book
2	201513138084	123456	0	维特无忧堡	test
3	123456	123456	0	孙悟空	shop
4	11	11	0	11	shop

image.png

1、select快照读（照片）

当你执行select *之后，在A与B事务中都会返回4条一样的数据，这是不用想的，当执行select的时候，innodb默认会执行快照读，相当于就是给你目前的状态找了一张照片，以后执行select 的时候就会返回当前照片里面的数据，当其他事务提交了也对你不造成影响，和你没关系，这就实现了可重复读了，那这个照片是什么时候生成的呢？不是开启事务的时候，是当你第一次执行select的时候，也就是说，当A开启了事务，然后没有执行任何操作，这时候B insert了一条数据然后commit,这时候A执行 select，那么返回的数据中就会有B添加的那条数据……之后无论再有其他事务commit都没有关系，因为照片已经生成了，而且不会再生成了，以后都会参考这张照片。

2、update、insert、delete 当前读

当你执行这几个操作的时候默认会执行当前读，也就是会读取最新的记录，也就是别的事务提交的数据你也可以看到，这样很好理解啊，假设你要update一个记录，另一个事务已经delete这条数据并且commit了，这样不是会产生冲突吗，所以你update的时候肯定要知道最新的信息啊。

我在这里介绍一下update的过程吧，首先会执行当前读，然后把返回的数据加锁，之后执行update。加锁是防止别的事务在这个时候对这条记录做什么，默认加的是排他锁，也就是你读都不可以，这样就可以保证数据不会出错了。但注意一点，就算你这里加了写锁，别的事务也还是能访问的，是不是很奇怪？数据库采取了一致性非锁定读，别的事务会去读取一个快照数据。

innodb默认隔离级别是RR， 是通过MVVC来实现了，读方式有两种，执行select的时候是快照读，其余是当前读，所以，mvcc不能根本上解决幻读的情况

MVCC

多版本并发控制

Multi-Version Concurrency Control 多版本并发控制，MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问；在编程语言中实现事务内存。

中文名	MVCC	被称为	多版本并发控制
增　加	并发性	避　免	使用锁
		保　存	某个时间点上的数据快照

目录

产品简介

大多数的MySQL事务型存储引擎，如InnoDB，Falcon以及PBXT都在使用一种简单的行锁机制。事实上，他们都和另外一种用来增加并发性的被称为“多版本并发控制（MVCC）”的机制来一起使用。MVCC不只使用在MySQL中，Oracle、PostgreSQL，以及其他一些数据库系统也同样使用它。

你可将MVCC看成行级别锁的一种妥协，它在许多情况下避免了使用锁，同时可以提供更小的开销。根据实现的不同，它可以允许非阻塞式读，在写操作进行时只锁定必要的记录。

MVCC会保存某个时间点上的数据快照。这意味着事务可以看到一个一致的数据视图，不管他们需要跑多久。这同时也意味着不同的事务在同一个时间点看到的同一个表的数据可能是不同的。如果你从来没有过这种体验的话，可能理解起来比较抽象，但是随着慢慢地熟悉这种理解将会很容易。

各个存储引擎对于MVCC的实现各不相同。这些不同中的一些包括乐观和悲观并发控制。我们将通过一个简化的InnoDB版本的行为来展示MVCC工作的一个侧面。

InnoDB：通过为每一行记录添加两个额外的隐藏的值来实现MVCC，这两个值一个记录这行数据何时被创建，另外一个记录这行数据何时过期（或者被删除）。但是InnoDB并不存储这些事件发生时的实际时间，相反它只存储这些事件发生时的系统版本号。这是一个随着事务的创建而不断增长的数字。每个事务在事务开始时会记录它自己的系统版本号。每个查询必须去检查每行数据的版本号与事务的版本号是否相同。让我们来看看当隔离级别是REPEATABLE READ时这种策略是如何应用到特定的操作的：

SELECT InnoDB必须每行数据来保证它符合两个条件：

1、InnoDB必须找到一个行的版本，它至少要和事务的版本一样老(也即它的版本号不大于事务的版本号)。这保证了不管是事务开始之前，或者事务创建时，或者修改了这行数据的时候，这行数据是存在的。

2、这行数据的删除版本必须是未定义的或者比事务版本要大。这可以保证在事务开始之前这行数据没有被删除。这里的不是真正的删除数据，而是标志出来的删除。真正意义的删除是在commit的时候。

符合这两个条件的行可能会被当作查询结果而返回。

INSERT：InnoDB为这个新行记录当前的系统版本号。

DELETE：InnoDB将当前的系统版本号设置为这一行的删除ID。

UPDATE：InnoDB会写一个这行数据的新拷贝，这个拷贝的版本为当前的系统版本号。它同时也会将这个版本号写到旧行的删除版本里。

这种额外的记录所带来的结果就是对于大多数查询来说根本就不需要获得一个锁。他们只是简单地以最快的速度来读取数据，确保只选择符合条件的行。这个方案的缺点在于存储引擎必须为每一行存储更多的数据，做更多的检查工作，处理更多的善后操作。

MVCC只工作在REPEATABLE READ和READ COMMITED隔离级别下。READ UNCOMMITED不是MVCC兼容的，因为查询不能找到适合他们事务版本的行版本；它们每次都只能读到最新的版本。SERIABLABLE也不与MVCC兼容，因为读操作会锁定他们返回的每一行数据 [1] 。