(九)MySQL之MVCC机制：为什么你改了的数据我还看不见？

竹子爱熊猫 LV.5
2022年10月17日 14:09 · 阅读 3825

引言

> 本文为掘金社区首发签约文章，14天内禁止转载，14天后未获授权禁止转载，侵权必究！

在《MySQL锁机制》这篇文章中，咱们全面剖析了`MySQL`提供的锁机制，对于并发事务通常可以通过其提供的各类锁，去确保各场景下的线程安全问题，从而能够防止脏写、脏读、不可重复读及幻读这类问题出现。

> 不过成也萧何败也萧何，虽然`MySQL`提供的锁机制确实能解决并发事务带来的一系列问题，但由于加锁后会让一部分事务串行化，而`MySQL`本身就是基于磁盘实现的，性能无法跟内存型数据库娉美，因此并发事务串行化会使其效率更低。

也正是由于上述原因，因此`MySQL`官方在设计时，抓破脑袋的想：**「有没有办法再快一点！」**！最终，`MVCC`机制就诞生了，相较于加锁串行化执行，`MVCC`机制的出现，则以另一种形式解决了并发事务造成的问题。

一、并发事务的四种场景

明，咱们首先来看看读-读场景。

1.1、读-读场景

读-读场景即是指多个事务/线程在一起读取一个相同的数据，比如事务 `T1` 正在读取 `ID=88` 的行记录，事务 `T2` 也在读取这条记录，两个事务之间是并发执行的。

> 广为人知的一点：`MySQL` 执行查询语句，绝对不会对引起数据的任何变化，因此对于这种情况而言，不需要做任何操作，因为不改变数据就不会引起任何并发问题。

1.2、写-写场景

写-写场景也比较简单，也就是指多个事务之间一起对同一数据进行写操作，比如事务 `T1` 对 `ID=88` 的行记录做修改操作，事务 `T2` 则对这条数据做删除操作，事务 `T1` 提交事务后想查询看一下，哦豁，结果连这条数据都不见了，这也是所谓的脏写问题，也被称为更新覆盖问题，对于这个问题在所有数据库、所有隔离级别中都是零容忍的存在，最低的隔离级别也要解决这个问题。

1.3、读-写、写-读场景

读-写、写-读实际上从宏观角度来看，可以理解成同一种类型的操作，但从微观角度而言则是两种不同的情况，读-写是指一个事务先开始读，然后另一个事务则过来执行写操作，写-读则相反，主要是读、写发生的前后顺序的区别。

> 并发事务中同时存在读、写两类操作时，这是最容易出问题的场景，脏读、不可重复读、幻读都出自于这种场景中，当有一个事务在做写操作时，读的事务中就有可能出现这一系列问题，因此数据库才会引入各种机制解决。

在《MySQL锁机制》中，对于写-写、读-写、写-读这三类场景，都是利用加锁的方案确保线程安全，但上面说到过，加锁会导致部分事务串行化，因此效率会下降，而 MVCC 机制的诞生则解决了这个问题。

> 先来设想一个问题：加锁的目的是什么？防止脏写、脏读、不可重复读及幻读这类问题出现。

对于脏写问题，这是写-写场景下会出现的，写-写场景必须要加锁才能保障安全，因此先将该场景排除在外。再想想：对于读-写并存的场景中，脏读、不可重复读及幻读问题都出自该场景中，但实际项目中，出现这些问题的几率本身就比较小，为了防止一些小概念事件，就将所有操纵同一数据的并发读写事务串行化，这似乎有些不讲道理呀，就好比：

> 为了防止自家保险柜中的 3.25 元被偷，所以每天从早到晚一直守着保险柜，这合理吗？并不合理，毕竟只有千日做贼，那有千日防贼的道理。

因此 MySQL 就基于读-写并存的场景，推出了 MVCC 机制，在线程安全问题和加锁串行化之间做了一定取舍，让两者之间达到了很好的平衡，即防止了脏读、不可重复读及幻读问题的出现，又无需对并发读-写事务加锁处理。

> 咋做到的呢？接下来一起来好好聊一聊大名鼎鼎的 MVCC 机制。

二、MySQL-MVCC机制综述

到即使有读写冲突时，也可以不加锁解决，从而确保了任何时刻的读操作都是非阻塞的。

> 但与其说是 MySQL-MVCC 机制，还不如说是 InnoDB-MVCC 机制，因为在 MySQL 众多的开源存储引擎中，几乎只有 InnoDB 实现了 MVCC 机制，类似于 MyISAM、Memory 等引擎中都未曾实现，那其他引擎为何不实现呢？不是不想，而是做不到，这跟 MVCC 机制的实现原理有关，这点放在后续详细讲解~

不过为了更好的理解啥叫 MVCC 多版本并发控制，先来看一个日常生活的例子~

2.1、MVCC技术在日常生活中的体现

不知道各位小伙伴中，是否有人做过论坛这类业务的项目，或者类似审核的业务需求，以掘金的文章为例，此时来思考一个场景：

> 假设我发布了一篇关于《MySQL事务机制》的文章，发布后挺受欢迎的，因此有不少小伙伴在看，其中有一位小伙伴比较细心，文中存在两三个错别字，被这位小伙伴指出来了，因此我去修正错别字后重新发布。

问题来了，对于文章首次发布也好，重新发布也罢，绝对要等审核通过后才会正式发布的，那我修正文章后重新发布，文章又会进入「审核中」这个状态，此时对于其他正在看、准备看的小伙伴来说，文章是不是就不见了？毕竟文章还在审核撒，因此对这个业务需求又该如何实现呢？多版本！

> 啥意思呢？也就是说，对于首次发布后通过审核的文章，在后续重新发布审核时，用户可以看到更新前的文章，也就是看到老版本的文章，当更新后的文章审核通过后，再使用新版本的文章代替老版本的文章即可。

而 MySQL-MVCC 机制的思想也大致相同。

2.2、MySQL-MVCC多版本并发控制

MySQL 中的多版本并发控制，也和上面给出的例子类似，毕竟回想一下，脏读、不可重复读、幻读问题都是由于多个事务并发读写导致的，但这些问题都是基于最新版本的数据并发操作才会出现，那如果读、写的事务操作的不是同一个版本呢？比如写操作走新版本，读操作走老版本，这样是不是无论执行写操作的事务干了啥，都不会影响读的事务？答案是 Yes 。

> 不过要稍微记住， MySQL 中仅在 RC 读已提交级别、 RR 不可重复读级别才会使用 MVCC 机制， Why ？

因为如果是 RU 读未提交级别，既然都允许存在脏读问题、允许一个事务读取另一个事务未提交的数据，那自然可以直接读最新版本的数据，因此无需 MVCC 介入。

同时如若是 Serializable 串行化级别，因为会将所有的并发事务串行化处理，也就是不论事务是读操作，亦或是写操作，都会被排好队一个个执行，这都不存在所谓的多线程并发问题了，自然也无需 MVCC 介入。

> 因此要牢记： MVCC 机制在 MySQL 中，仅有 InnoDB 引擎支持，而在该引擎中， MVCC 机制只对 RC、RR 两个隔离级别下的事务生效。当然， RC、RR 两个不同的隔离级别中， MVCC 的实现也存在些许差异，对于这点后续详细讲解。

三、MySQL-MVCC机制实现原理剖析

OK~，简单理解了啥叫 MVCC 机制后，接着一起来看看 InnoDB 引擎是如何实现它的， MVCC 机制主要通过隐藏字段、 Undo-log 日志、 ReadView 这三个东西实现的，用两这三玩

通常而言，当你基于 InnoDB 引擎建立一张表后， MySQL 除开会构建你显式声明的字段外，通常还会构建一些 InnoDB 引擎的隐藏字段，在 InnoDB 引擎中主要有 DB_ROW_ID、DB_Deleted_Bit、DB_TRX_ID、DB_ROLL_PTR 这四个隐藏字段，挨个简单介绍一下。

3.1.1、隐藏主键 - ROW_ID（6Bytes）

在之前介绍 《索引原理篇》 的时候聊到过一点，对于 InnoDB 引擎的表而言，由于其表数据是按照聚簇索引的格式存储，因此通常都会选择主键作为聚簇索引列，然后基于主键字段构建索引树，但如若表中未定义主键，则会选择一个具备唯一非空属性的字段，作为聚簇索引的字段来构建树。

> 当两者都不存在时， InnoDB 就会隐式定义一个顺序递增的列 ROW_ID 来作为聚簇索引列。

因此要牢记一点，如果你选择的引擎是 InnoDB ，就算你的表中未定义主键、索引，其实默认也会存在一个聚簇索引，只不过这个索引在上层无法使用，仅提供给 InnoDB 构建树结构存储表数据。

3.1.2、删除标识 - Deleted_Bit（1Bytes）

在之前讲 《SQL执行篇-写SQL执行原理》 时，咱们只粗略的过了一下大体流程，其中并未涉及到一些细节阐述，在这里稍微提一下：对于一条 delete 语句而言，当执行后并不会立马删除表的数据，而是将这条数据的 Deleted_Bit 删除标识改为 1/true ，后续的查询 SQL 检索数据时，如果检索到了这条数据，但看到隐藏字段 Deleted_Bit=1 时，就知道该数据已经被其他事务 delete 了，因此不会将这条数据纳入结果集。

> OK~，但设计 `Deleted_Bit` 这个隐藏字段的好处是什么呢？主要是能够有利于聚簇索引，比如当一个事务中删除一条数据后，后续又执行了回滚操作，假设此时是真正的删除了表数据，会发生什么情况呢？

- ①删除表数据时，有可能会破坏索引树原本的结构，导致出现叶子节点合并的情况。
- ②事务回滚时，又需重新插入这条数据，再次插入时又会破坏前面的结构，导致叶子节点分裂。

> 综上所述，如果执行 `delete` 语句就删除真实的表数据，由于事务回滚的问题，就很有可能导致聚簇索引树发生两次结构调整，这其中的开销可想而知，而且先删除，再回滚，最终树又变成了原状，那这两次树的结构调整还是无意义的。

所以，当执行 `delete` 语句时，只会改变将隐藏字段中的删除标识改为 `1/true` ，如果后续事务出现回滚动作，直接将其标识再改回 `0/false` 即可，这样就避免了索引树的结构调整。

> 但如若事务删除数据之后提交了事务呢？总不能让这条数据一直留在磁盘吧？毕竟如果所有的 `delete` 操作都这么干，就会导致磁盘爆满~，显然这样是不妥的，因此删除标识为 `1/true` 的数据最终依旧会从磁盘中移除，啥时候移呢？

在之前讲 《Nginx-缓存清理》 时，曾经提到过 `purger` 这一系列的参数，通过配置该系列参数后， `Nginx` 后台中会创建对应的 `purger` 线程去自动删除缓存数据。而 `MySQL` 中也不例外，同样存在 `purger` 线程的概念，为了防止“已删除”的数据占用过多的磁盘空间， `purger` 线程会自动清理 `Deleted_Bit=1/true` 的行数据。

> 当然，为了确保清理数据时不会影响 `MVCC` 的正常工作， `purger` 线程自身也会维护一个 `ReadView` ，如果某条数据的 `Deleted_Bit=true` ，并且 `TRX_ID` 对 `purge` 线

>

对于上述最后一段大家可能会有些许疑惑，这是因为还未曾介绍 ReadView，因此有些不理解可先跳过，后续理解了 ReadView 后再回来看会好很多。

3.1.3、最近更新的事务ID - TRX_ID（6Bytes）

TRX_ID 全称为 transaction_id，翻译过来也就是事务 ID 的意思，MySQL 对于每一个创建的事务，都会为其分配一个事务 ID，事务 ID 同样遵循顺序递增的特性，即后来的事务 ID 绝对会比之前的 ID 要大，比如：

> 此时事务 T1 准备修改表字段的值，MySQL 会为其分配一个事务 ID=1，当事务 T2 准备向表中插入一条数据时，又会为这个事务分配一个 ID=2 .....

但有一个细节点需要记住：MySQL 对于所有包含写入 SQL 的事务，会为其分配一个顺序递增的事务 ID，但如果是一条 select 查询语句，则分配的事务 ID=0。

> 不过对于手动开启的事务，MySQL 都会为其分配事务 ID，就算这个手动开启的事务中仅有 select 操作。

表中的隐藏字段 TRX_ID，记录的就是最近一次改动当前这条数据的事务 ID，这个字段是实现 MVCC 机制的核心之一。

3.1.4、回滚指针 - ROLL_PTR（7Bytes）

log 日志中，而 rollback_pointer 就是一个地址指针，指向 Undo-log 日志中旧版本的数据，当需要回滚事务时，就可以通过这个隐藏列，来找到改动之前的旧版本数据，而 MVCC 机制也利用这点，实现了行数据的多版本。

3.2、InnoDB引擎的Undo-log日志

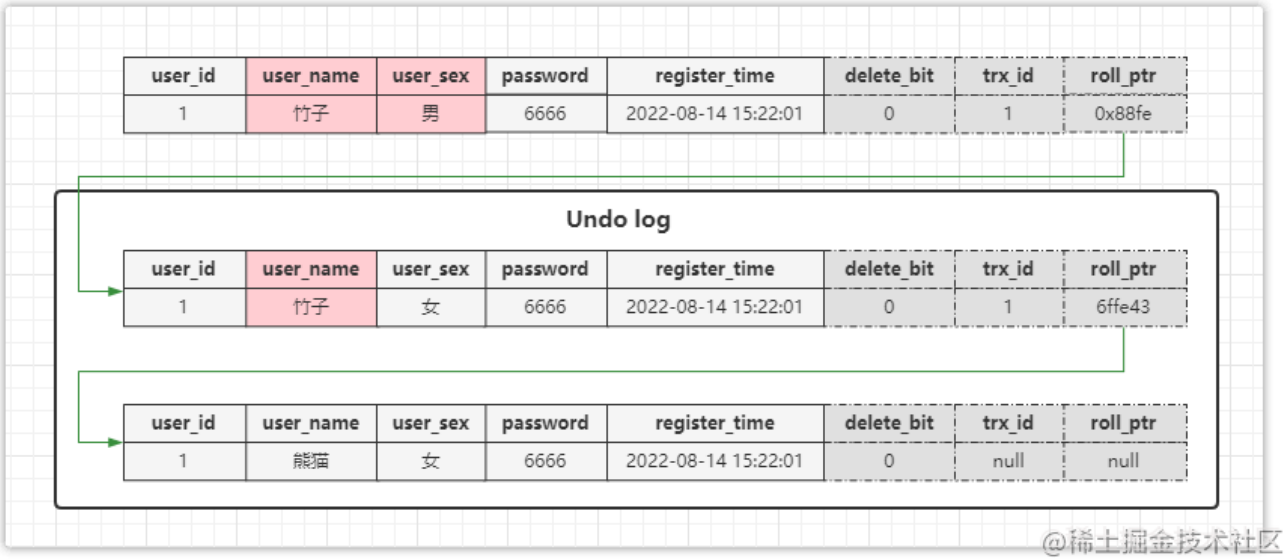
在之前《事务篇》中分析事务实现原理时，咱们得知了 MySQL 事务机制是基于 Undo-log 实现的，同时在刚刚在聊回滚指针时，聊到了 Undo-log 日志中会存储旧版本的数据，但要注意：Undo-log 中并不仅仅只存储一条旧版本数据，其实在该日志中会有一个版本链，啥意思呢？举个例子：

sql 复制代码

```sql
SELECT * FROM `zz_users` WHERE user_id = 1;
+---------+-----------+----------+----------+---------------------+
| user_id | user_name | user_sex | password | register_time       |
+---------+-----------+----------+----------+---------------------+
|       1 | 熊猫      | 女       | 6666     | 2022-08-14 15:22:01 |
+---------+-----------+----------+----------+---------------------+

UPDATE `zz_users` SET user_name = "竹子" WHERE user_id = 1;
UPDATE `zz_users` SET user_sex = "男" WHERE user_id = 1;
```

比如上述这段 SQL 隶属于 trx_id=1 的 T1 事务，其中对同一条数据改动了两次，那 Undo-log 日志中只会存储一条旧版本数据吗？NO，答案是两条旧版本的数据，如下图：



的旧版本数据组成一个单向链表。但要注意一点：最新的旧版本数据，都会插入到链表头中，而不是追加到链表尾部。

> 细说一下执行上述 `update` 语句的详细过程：
> ①对 `ID=1` 这条要修改的行数据加上排他锁。
> ②将原本的旧数据拷贝到 `Undo-log` 的 `rollback Segment` 区域。
> ③对表数据上的记录进行修改，修改完成后将隐藏字段中的 `trx_id` 改为当前事务 `ID` 。
> ④将隐藏字段中的 `roll_ptr` 指向 `Undo-log` 中对应的旧数据，并在提交事务后释放锁。

为什么 `Undo-log` 日志要设计出版本链呢？两个好处：一方面可以实现事务点回滚（这点回去参考事务篇），另一方面则可以实现 `MVCC` 机制（这点后面聊）。

> 与之前的删除标识类似，一条数据被 `delete` 后并提交了，最终会从磁盘移除，而 `Undo-log` 中记录的旧版本数据，同样会占用空间，因此在事务提交后也会移除，移除的工作同样由 `purger` 线程负责， `purger` 线程内部也会维护一个 `ReadView` ，它会以此作为判断依据，来决定何时移除 `Undo` 记录。

3.3、MVCC核心 - ReadView

`MVCC` 在前面聊到过，它翻译过来就是多版本并发控制的意思，对于这个名词中的多版本已经通过 `Undo-log` 日志实现了，但再思考一个问题：如果 `T2` 事务要查询一条行数据，此时这条行数据正在被 `T1` 事务写，那也就代表着这条数据可能存在多个旧版本数据， `T2` 事务在查询时，应该读这条数据的哪个版本呢？此时就需要用到 `ReadView` ，用它来做多版本的并发控制，根据查询的时机来选择一个当前事务可见的旧版本数据读取。

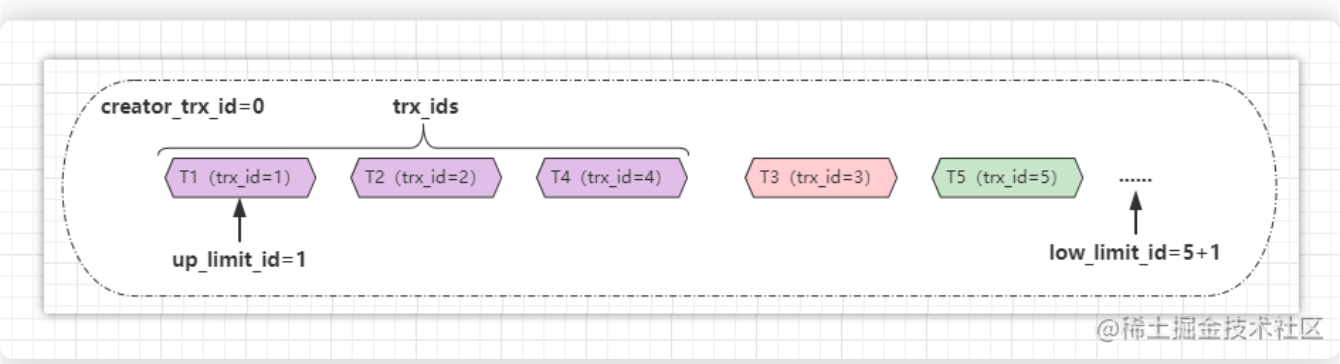
> 那究竟什么是 ReadView 呢？就是一个事务在尝试读取一条数据时， MVCC 基于当前 MySQL 的运行状态生成的快照，也被称之为读视图，即 ReadView ，在这个快照中记录者当前所有活跃事务的 ID （活跃事务是指还在执行的事务，即未结束（提交/回滚）的事务）。

当一个事务启动后，首次执行 select 操作时， MVCC 就会生成一个数据库当前的 ReadView ，通常而言，一个事务与一个 ReadView 属于一对一的关系（不同隔离级别下也会存在细微差异）， ReadView 一般包含四个核心内容：

- creator_trx_id ：代表创建当前这个 ReadView 的事务 ID 。
- trx_ids ：表示在生成当前 ReadView 时，系统内活跃的事务 ID 列表。
- up_limit_id ：活跃的事务列表中，最小的事务 ID 。
- low_limit_id ：表示在生成当前 ReadView 时，系统中要给下一个事务分配的 ID 值。

上面四个值很简单，值得一提的是 low_limit_id ，它并不是目前系统中活跃事务的最大 ID ，因为之前讲到过， MySQL 的事务 ID 是按序递增的，因此当启动一个新的事务时，都会为其分配事务 ID ，而这个 low_limit_id 则是整个 MySQL 中，要为下一个事务分配的 ID 值。

下面上个 ReadView 的示意图，来好好理解一下它：



假设目前数据库中共有 T1~T5 这五个事务， T1、T2、T4 还在执行， T3 已经回滚， T5 已经提交，此时当有一条查询语句执行时，就会利用 MVCC 机制生成一个 ReadView ，由于前面讲过，单纯由一条 select 语句组成的事务并不会分配事务 ID ，因此默认为 0 ，所以目前这个快照的信息如下：

```
json 复制代码
```

```
}
```

OK~，简单明白 ReadView 的结构后，接着一起来聊一聊 MVCC 机制的实现原理。

3.4、MVCC机制实现原理

将“ MVCC 三剑客”的概念阐述完毕后，再结合三者来谈谈 MVCC 的实现，其实也比较简单，经过前面的讲解后已得知：

- ①当一个事务尝试改动某条数据时，会将原本表中的旧数据放入 Undo-log 日志中。
- ②当一个事务尝试查询某条数据时， MVCC 会生成一个 ReadView 快照。

其中 Undo-log 主要实现数据的多版本， ReadView 则主要实现多版本的并发控制，还是以之前的例子来举例说明：

sql 复制代码

```sql
-- 事务T1：trx_id=1
UPDATE `zz_users` SET user_name = "竹子" WHERE user_id = 1;
UPDATE `zz_users` SET user_sex = "男" WHERE user_id = 1;
```

sql 复制代码

```sql
-- 事务T2：trx_id=2
SELECT * FROM `zz_users` WHERE user_id = 1;
```

目前存在 T1、T2 两个并发事务， T1 目前在修改 ID=1 的这条数据，而 T2 则准备查询这条数据，那么 T2 在执行时具体过程是怎么回事呢？如下：

- ①当事务中出现 select 语句时，会先根据 MySQL 的当前情况生成一个 ReadView 。
- ②判断行数据中的隐藏列 trx_id 与 ReadView.creator_trx_id 是否相同：
  - 相同：代表创建 ReadView 和修改行数据的事务是同一个，自然可以读取最新版数据。
  - 不相同：代表目前要查询的数据，是被其他事务修改过的，继续往下执行。
- ③判断隐藏列 trx_id 是否小于 ReadView.up_limit_id 最小活跃事务 ID ：
  - 小于：代表改动行数据的事务在创建快照前就已结束，可以读取最新版本的数据。
  - 不小于：则代表改动行数据的事务还在执行，因此需要继续往下判断。
- ④判断隐藏列 trx_id 是否小于 ReadView.low_limit_id 这个值：

- ⑤如果隐藏列 `trx_id` 小于 `low_limit_id`，继续判断 `trx_id` 是否在 `trx_ids` 中：
  - 在：表示改动行数据的事务目前依旧在执行，不能访问最新版数据。
  - 不在：表示改动行数据的事务已经结束，可以访问最新版的数据。

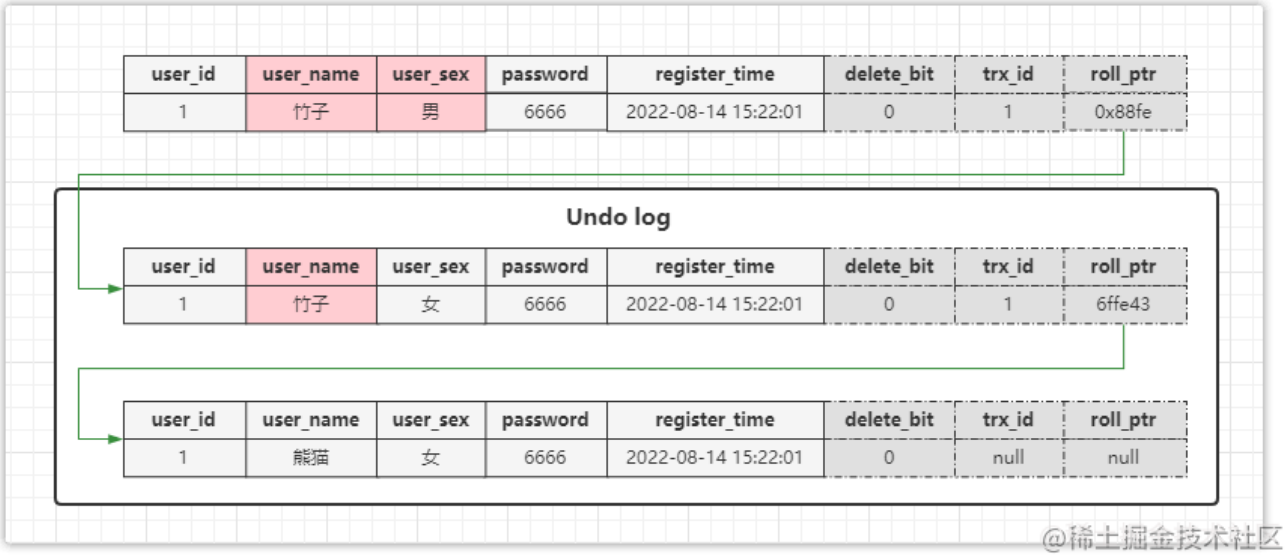
说简单一点，就是首先会去获取表中行数据的隐藏列，然后经过上述一系列判断后，可以得知：**「目前查询数据的事务到底能不能访问最新版的数据」**。如果能，就直接拿到表中的数据并返回，反之，不能则去 `Undo-log` 日志中获取旧版本的数据返回。

> 注意：假设 `Undo-log` 日志中存在版本链怎么办？该获取哪个版本的旧数据呢？

如果 `Undo-log` 日志中的旧数据存在一个版本链时，此时会首先根据隐藏列 `roll_ptr` 找到链表头，然后依次遍历整个列表，从而检索到最合适的一条数据并返回。但在这个遍历过程中，是如何判断一个旧版本的数据是否合适的呢？条件如下：

- 旧版本的数据，其隐藏列 `trx_id` 不能在 `ReadView.trx_ids` 活跃事务列表中。

因为如果旧版本的数据，其 `trx_id` 依旧在 `ReadView.trx_ids` 中，就代表着产生这条旧数据的事务还未提交，自然不能读取这个版本的数据，以前面给出的例子来说明：



这是由事务 `T1` 生成的版本链，此时 `T2` 生成的 `ReadView` 如下：

```
    "trx_ids" : "[1]",
    "up_limit_id" : "1",
    "low_limit_id" : "2"
}
```

结合这个 `ReadView` 信息，经过前面那一系列判断后，最终会得到：不能读取最新版数据，因此需要去 `Undo-log` 的版本链中读数据，首先根据 `roll_ptr` 找到第一条旧数据：

user_id	user_name	user_sex	password	register_time	delete_bit	trx_id	roll_ptr
1	竹子	女	6666	2022-08-14 15:22:01	0	1	6ffe43

此时发现其 `trx_id=1`，位于 `ReadView.trx_ids` 中，因此不能读取这条旧数据，接着再根据这条旧数据的 `roll_ptr` 找到第二条旧版本数据：

user_id	user_name	user_sex	password	register_time	delete_bit	trx_id	roll_ptr
1	熊猫	女	6666	2022-08-14 15:22:01	0	null	null

这时再看其 `trx_id=null`，并不位于 `ReadView.trx_ids` 中，`null` 表示这条数据在上次 `MySQL` 运行时就已插入了，因此这条旧版本的数据可以被 `T2` 事务读取，最终 `T2` 就会查询到这条数据并返回。

> OK~，最后再来看一个场景！即范围查询时，突然出现新增数据怎么办呢？如下：

sql 复制代码

```
SELECT * FROM `zz_users`;
+---------+-----------+---------+----------+---------------------+
| user_id | user_name | user_sex | password | register_time       |
+---------+-----------+---------+----------+---------------------+
|       1 | 熊猫      | 女       | 6666     | 2022-08-14 15:22:01 |
```

```sql
-- T1事务：查询ID >= 3 的所有用户信息
select * from  `zz_users` where user_id >= 3;

-- T2事务：新增一条 ID = 6 的用户记录
INSERT INTO `zz_users` VALUES(6,"棕熊","男","7777","2022-10-02 16:21:33");
```

此时当 T1 事务查询数据时，突然蹦出来一条 ID=6 的数据，经过判断之后会发现新增这条数据的事务还在执行，所以要去查询旧版本数据，但此时由于是新增操作，因此 roll_ptr=null ，即表示没有旧版本数据，此时会不会读取最新版的数据呢？答案是 NO ，如果查询数据的事务不能读取最新版数据，同时又无法从版本链中找到旧数据，那就意味着这条数据对 T1 事务完全不可见，因此 T1 的查询结果中不会包含 ID=6 的这条新增记录。

3.5、RC、RR不同级别下的MVCC机制

3.4 阶段已经将 MVCC 机制的具体实现过程剖析了一遍，接下来再思考一个问题：

> ReadView 是一个事务中只生成一次，还是每次 select 时都会生成呢？

这个问题的答案跟事务的隔离机制有关，不同级别的隔离机制也并不同，如果此时 MySQL 的事务隔离机制处于 RC 读已提交级别，那此时来看一个例子：

sql 复制代码

```sql
-- 开启一个事务T1：主要是修改两次ID=1的行数据
begin;
UPDATE `zz_users` SET user_name = "竹子" WHERE user_id = 1;
UPDATE `zz_users` SET user_sex = "男" WHERE user_id = 1;

-- 再开启一个事务T2：主要是查询ID=1的行数据
SELECT * FROM `zz_users` WHERE user_id = 1;

-- 此时先提交事务T1
commit;
```

> 先说明一点，为了方便理解，因此我将两个事务的代码贴在了一块，但如若你要做实际的实验，请切记将 `T1、T2` 用两个连接来写。

OK~，再来看看上述这个案例，如果是处于 `RC` 级别的情况下， `T2` 事务中的查询结果如下：

sql 复制代码

```
SELECT * FROM `zz_users` WHERE user_id = 1;
+---------+-----------+----------+----------+---------------------+
| user_id | user_name | user_sex | password | register_time       |
+---------+-----------+----------+----------+---------------------+
|       1 | 熊猫      | 女       | 6666     | 2022-08-14 15:22:01 |
+---------+-----------+----------+----------+---------------------+

SELECT * FROM `zz_users` WHERE user_id = 1;
+---------+-----------+----------+----------+---------------------+
| user_id | user_name | user_sex | password | register_time       |
+---------+-----------+----------+----------+---------------------+
|       1 | 竹子      | 男       | 6666     | 2022-08-14 15:22:01 |
+---------+-----------+----------+----------+---------------------+
```

为什么两次查询结果不一样呢？因为 `RC` 级别下， `MVCC` 机制是会在每次 `select` 语句执行前，都会生成一个 `ReadView` ，由于 `T2` 事务中第二次查询数据时， `T1` 已经提交了，所以第二次查询就能读到修改后的数据，这是啥问题？不可重复读问题。

> 接着再来看看 `RR` 可重复级别下的 `MVCC` 机制， `SQL` 代码和上述一模一样，但查询结果如下：

sql 复制代码

```
SELECT * FROM `zz_users` WHERE user_id = 1;
+---------+-----------+----------+----------+---------------------+
| user_id | user_name | user_sex | password | register_time       |
+---------+-----------+----------+----------+---------------------+
```

```
+---------+-----------+----------+----------+---------------------+
| user_id | user_name | user_sex | password | register_time       |
+---------+-----------+----------+----------+---------------------+
|       1 | 熊猫      | 女       | 6666     | 2022-08-14 15:22:01 |
+---------+-----------+----------+----------+---------------------+
```

这又是为啥？为啥明明在 T2 事务第二次查询前， T1 已经提交了， T2 依旧查询出的结果和第一次相同呢？这是因为在 RR 级别中，一个事务只会在首次执行 select 语句时生成快照，后续所有的 select 操作都会基于这个 ReadView 来判断，这样也就解决了 RC 级别中存在的不可重复问题。

> 最后简单提一嘴：实际上 InnoDB 引擎中，是可以在 RC 级别解决脏读、不可重复读、幻读这一系列问题的，但是为了将事务隔离级别设计的符合 DBMS 规范，因此在实现时刻意保留了这些问题，然后放在更高的隔离级别中解决~

四、MVCC机制篇总结

MVCC 多版本并发控制，听起来似乎蛮高大上的，但实际研究起来会发现它并不复杂，其中的多版本主要依赖 Undo-log 日志来实现，而并发控制则通过表的隐藏字段+ ReadView 快照来实现，通过 Undo-log 日志、隐藏字段、 ReadView 快照这三玩意儿，就实现了 MVCC 机制，过程还蛮简单的~

> 到这里，其实对于 MySQL 的事务隔离机制，已经拨开一部分迷雾了，下篇《MySQL事务与锁机制原理篇》中，则会彻底讲清楚 MySQL 锁是怎么实现的，以及不同的事务隔离级别，又是如何借助锁+ MVCC 处理客户端 SQL 的，那么咱们下篇见~

分类： 后端 标签： 数据库 MySQL 面试

全解MySQL数据库

从MySQL整体架构出发，到SQL优化、MySQL索引、慢查询优化...

关注专栏

相关课程

分布式服务面试精讲

天涯兰

774购买

¥49.9

VIP 玩转 MyBatis：深度解析与定制

LinkedBear LV.3

1710购买

¥49.9

评论

输入评论（Enter换行，Ctrl + Enter发送）

全部评论 63

最新 最热

用户881219210... JY.3

14天前

这写得也太棒了吧我淦 😍

点赞 1

竹子爱熊猫 （作者）

14天前

感谢认可 🥰

点赞 回复

小小录 JY.3 研究生 @ 浙江大学

15天前

71 63 收藏

竹子爱熊猫 （作者） 14天前

insert插入呀😂，表数据的写操作有三类：增、删、改，删和改都是基于有数据的情况下进行的，但增操作是基于无数据的情况下进行的，所以insert之前不存在数据，因此roll_ptr会为null，这样不知道你是否理解呢

点赞 回复

小小录 回复 竹子爱熊猫 5天前

嗯嗯理解了，谢谢作者，写的真棒

> “insert插入呀😂，表数据的写操作有三类：增、删、改，删和改都是基于有数... ”

点赞 回复

查看更多回复 ∨

用户573675446... JY.3 19天前

过程写的通俗易懂

点赞 1

竹子爱熊猫 （作者） 19天前

三克油~，感谢认可🥰

点赞 回复

用户900357734... JY.2 20天前

👍

点赞 1

竹子爱熊猫 （作者） 20天前

🥰

点赞 回复

用户422277033... JY.4 24天前

71 63 收藏

的事物，因为是头插法怎么保证在rr 隔离级别下每次遍历出来的都是同一个版本，因为有可能多次读的场景下trxids在不断减少

点赞 2

竹子爱熊猫 （作者） 23天前

第一个问题的话，你可以看一下readview的定义，参考我下面的截图，trx_ids是存放生成快照时，系统内所有活跃的事务ID列表，不是指当前事务。

点赞 回复

竹子爱熊猫 （作者） 23天前

第二个问题：RR级别下，一个事务内只有第一次读的时候，才会生成一个快照，后续所有的读操作，都会基于第一次生成的快照去做MVCC读数据，所以就能够确保读到的数据是一致的，trx_ids列表中的事务ID是不会变化的，只有在RC级别下才会每次select都生成快照。

点赞 回复

颜如玉 LV.3 JY.5 Java混子 @ 野生技术协... 24天前

原文：①当一个事务尝试改动某条数据时，会将原本表中的旧数据放入Undo-log日志中。
作者你好，请教一下：这里的旧数据是指完整的那条记录，还是那条记录中会被修改的字段原值？通过roll_ptr找到undo log链，比较trx_id如果可见就返回了，这中间若是没有其他动作，应该存的就是原先的那一整条记录，这样会多占用一些空间但是实现会简单很多

点赞 5

竹子爱熊猫 （作者） 24天前

完整数据哈，因为如果只放变更过的数据，则还需要额外的变量来记录是哪个字段发生了改变，同时在做事务回滚时，就无法直接覆盖，而是需要生成真正的反SQL语句执行做补偿，这种效率会比较低，因此会存储完整的行记录。

点赞 回复

颜如玉 回复 竹子爱熊猫 24天前

熊猫老板可以试试fancy这个主题，我感觉这个是掘金最漂亮的主题

71 63 收藏

查看更多回复 ⌄

用户356444964... JY.3 25天前

大佬，【几乎只有InnoDB实现了MVCC机制，类似于MyISAM、Memory等引擎中都未曾实现，那其他引擎为何不实现呢？不是不想，而是做不到，这跟MVCC机制的实现原理有关】能不能具体说一下为啥做不到？🤣，看完之后感觉明白了，到总结的时候咋还有点茫然那，哈哈

点赞 6

竹子爱熊猫 （作者） 25天前

因为MVCC机制的三大核心：隐藏字段（事务ID+回滚指针）、Undo-log日志的版本链、ReadView读视图。

目前MySQL的常用引擎中，基本上只有InnoDB支持事务，所以只有InnoDB的表上才会有隐藏的事务ID。
同时Undo-log日志是InnoDB引擎专属的，所以也只有InnoDB表才有回滚指针，以及...

展开

点赞 回复

竹子爱熊猫 （作者） 25天前

这样跟你去解释应该会清晰很多~，还有其他疑惑也可以随时继续留言啦！

点赞 回复

查看更多回复 ⌄

xiaobinqt JY.3 26天前

请教一个问题，这 2 个 `trx_id` 为什么是一样的？感谢。

点赞 1

竹子爱熊猫 （作者） 26天前

因为你仔细看一下前面给出的例子，我是在T1事务中对同一条数据修改了两次，所以生

71 63 收藏

点赞 回复

xiaobinqt JY.3 26天前

好像有个错别字，抑或？

点赞 1

竹子爱熊猫 （作者） 26天前

亦或，是要打这个词的，我找一下哈，修改一下，敲字敲多了就打错了

点赞 回复

YesNoBut LV.1 JY.4 Java 1月前

大佬，请教个问题，在3.4小结中，T2的具体执行过程描述中，ReadView和行数据比较，这里的行数据读取的是哪里的，磁盘还是undo-log中的数据?

点赞 4

竹子爱熊猫 （作者） 1月前

具体是那块呢，你可以截图出来给我看看，我3.4阶段没写小结撒

点赞 回复

竹子爱熊猫 （作者） 1月前

你是指这块的区域嘛？ReadView中对比行数据时，会首先读取表中的最新数据，然后判断隐藏的事务ID字段，从而来得知这条数据是否是当前事务可读的，如果不可读则一直根据roll_ptr回滚指针，去找undo-log日志中的旧数据，持续进行比较。

...

展开

71 63 收藏

失态 JY.3 1月前

收藏+关注 学到了学到了

点赞 4

竹子爱熊猫 （作者） 1月前

哈哈哈，但也要认真读一读，真正的看完了，才能变成自己的🧐。

点赞 回复

失态 回复 竹子爱熊猫 1月前

那是当然😁

> “哈哈哈，但也要认真读一读，真正的看完了，才能变成自己的🧐。”

点赞 回复

查看更多回复 ∨

宁在春 LV.5 JY.6 @ Java 1月前

今天我才来看，竹哥，我不晚吧，哈哈😽三连再看 ~

点赞 5

竹子爱熊猫 （作者） 1月前

欢迎~，不晚不晚哈哈哈，最近关注越来越多，点赞越来越少了，感谢大佬三连🧐

1 回复

宁在春 回复 竹子爱熊猫 1月前

我竹哥是个宝藏作者，可能是他们还没跟上你写的步伐，还在前头慢慢看勒~ 今天下午打算看第十章啦~ 学到了很多知识，虽然工作中我还用不到，但是思路和眼界都被打开啦~🥰

> “欢迎~，不晚不晚哈哈哈，最近关注越来越多，点赞越来越少了，感谢大佬三连... ”

点赞 回复

查看更多回复 ∨

71 63 收藏

MySQL大约有多少专栏，到时候打印PDF看纸质书

点赞 1

竹子爱熊猫 （作者） 1月前

那就多了😂，预估是30章左右，总计下来应该接近三十万字，打印出来估计得花不少钱，而且里面有动图，你打印的时候没办法打印动图的。

点赞 回复

StoneDB LV.2 JY.5 数据库架构师 @ 石原... 1月前

发现一个宝藏博主，更新的更的比我读的都快😭

2 1

竹子爱熊猫 （作者） 1月前

夸张了夸张了😂

点赞 回复

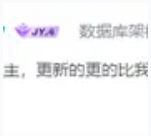
用户529970381... JY.2 1月前

发现一个宝藏博主，更的比我读的都快

点赞 1

竹子爱熊猫 （作者） 1月前

夸张了🤦，楼上有位大佬跟你节奏了🧐



1 回复

codeLeon JY.4 前端开发工程师 1月前

一个前端看你写的MySQL，文章质量太高了，膜拜👍

点赞 3

71 63 收藏

点赞 回复

codeLeon 回复 竹子爱熊猫 1月前

全不会工程师😂

> “大前端嘛，还是“全干型”前端🐱”

点赞 回复

查看更多回复

懿 JY.5 后端开发 @ 小白公司 1月前

😠😠😠

点赞 1

竹子爱熊猫 （作者） 1月前

欢迎大佬光临🫣

点赞 回复

CC_LKL JY.4 1月前

因为mysql用了MVCC，是不是其他事务提交的数据，在rc级别下，一个读的事务照样看不见啊🧐

点赞 3

竹子爱熊猫 （作者） 1月前

RC级别下的这个要分情况，好比写数据的事务叫T1，读数据的事务叫T2。
如果T1先开始写，并且在T2第一次读之前提交了事务，那么T1写的数据T2就能看到。
但如果T2先开始读，T1再开始写，那么T1写的数据T2就看不到。
还有一种情况，T1先开始写，T2再开始读，但T1在T2第一次读之前还没提交事务，T1写的数据，T2也是看不到的。
...

展开

1 回复

CC_LKL 回复 竹子爱熊猫 1月前

查看更多回复 ⌄

CC_LKL JY.4 1月前

持续追更😎

点赞 回复

相关推荐

一灯架构 | 3月前 | 后端 · 数据库 · MySQL

我说MySQL联合索引遵循最左前缀匹配原则，面试官让我回去等通知

1.8w 160 37

摸鱼的春哥 | 10月前 | 前端 · JavaScript

2022，前端的天🌩要怎么变?

17.6w 1698 733

杰出D | 1年前 | 前端 · 算法

面试了十几个高级前端，竟然连（扁平数据结构转Tree）都写不出来

24.4w 3572 1871

Gaby | 1年前 | JavaScript · 面试

🔥 连八股文都不懂还指望在前端混下去么

53.8w 10810 553

阳光是sunny | 8月前 | 前端 · JavaScript

三面面试官：运行 npm run xxx 的时候发生了什么?

14.5w 4634 405

前端阿飞 | 1年前 | 前端 · JavaScript

10个常见的前端手写功能，你全都会吗?

14.3w 3503 314

579 5 评论

大帅老猿 | 1年前 | 前端 · JavaScript

产品经理：你能不能用div给我画条龙?

12.8w 3081 627

程序员依扬 | 3年前 | 面试 · 前端

【1 月最新】前端 100 问：能搞懂 80% 的请把简历给我

59.7w 10432 356

荒山 | 3年前 | 前端 · 团队管理

if 我是前端团队 Leader，怎么制定前端协作规范?

17.2w 4503 228

码个蛋 | 5年前 | Android · RxJava · Kotlin

花了 4 个月整理了 50 篇 Android 干货文章

4.1w 2584 47

Sunshine_Lin | 1年前 | 前端 · Vue.js · 面试

后端一次给你10万条数据，如何优雅展示，到底考察我什么?

12.1w 1490 253

zz | 1年前 | 前端 · 面试

后端一次给你10万条数据,如何优雅展示，面试官到底考察我什么?

19.6w 545 333

神三元 | 2年前 | JavaScript · 面试

（建议精读）HTTP灵魂之问，巩固你的 HTTP 知识体系

21.7w 4707 178

黄小虫 | 2年前 | JavaScript

2020年了,再不会webpack敲得代码就不香了(近万字实战)

17.9w 4580 385

6.5w 668 140

~藕爸~ | 4年前 | 数据库 · 后端 · 架构

10亿级订单系统分库分表设计思路！

2.4w 429 22

vortesnail | 9月前 | 前端 · 面试

做了一份前端面试复习计划，保熟～

37.6w 7225 384

非优秀程序员 | 1年前 | 前端 · JavaScript

如何用 CSS 中写出超级美丽的阴影效果

105.0w 870 55