

## (十)全解MySQL之死锁问题分析、事务隔离与锁机制的底层原理剖析

竹子爱熊猫 LV.5

2022年10月19日 14:49 · 阅读 3084

✓ 已关注



### 📖 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！

”

经过 [《MySQL锁机制》](#)、[《MySQL-MVCC机制》](#) 两篇后，咱们已经大致了解 MySQL 中处理并发事务的手段，不过对于锁机制、MVCC 机制都并未与之前说到的 [《MySQL事务机制》](#) 产生关联关系，同时对于 MySQL 锁机制的实现原理也未曾剖析，因此本篇作为事务、锁、MVCC 这三者的汇总篇，会在本章中补全之前空缺的一些细节，同时也会将锁、MVCC 机制与事务机制之间的关系彻底理清楚。

### 📖 一、MySQL中的死锁现象

还记得咱们在 [《MySQL锁机制》](#) 这篇文章中，描述事务、连接、线程三者关系的那段话嘛？

客户端发往 MySQL 的一条条 SQL 语句，实际上都可以理解成一个个单独的事务，而在前面的 [《MySQL事务篇》](#) 中提到过：事务是基于数据库连接的，而每个数据库连接在 MySQL 中，又会用一条工作线程来维护，也意味着一个事务的执行，本质上就是一条工作线程在执行，当出现多个事务同时执行时，这种情况则被称之为并发事务，所谓的并发事务也就是指多条线程并发执行。

@稀土掘金技术社区

所谓的并发事务，本质上就是 MySQL 内部多条工作线程并行执行的情况，也正由于 MySQL 是多线程应用，所以需要具备完善的锁机制来避免线程不安全问题的产生，但熟悉多线程编程的小伙伴应该都清楚一点，对于多线程与锁而言，存在一个 100% 会出现的偶发问题，「即死锁问题」。

#### 🔗 1.1、死锁问题概述 (Dead Lock) 🔗

对于死锁的定义，这里就不展开叙述了，因为在之前 [《并发编程-死锁、活锁、锁饥饿》](#) 中曾详细描述过，如下：

## 二、死锁、活锁与锁饥饿概念理解

2.1、何谓死锁 (DeadLock) ?

2.2、活锁 (LiveLock) 是什么?

活锁解决方案

2.3、啥又叫锁饥饿 (LockStarving...)

## 三、死锁产生原因/如何避免死锁、排...

3.1、死锁产生的四个必要条件

3.2、系统资源的分类

3.2.1、永久性资源

3.2.2、临时性资源

3.2.3、可抢占式资源

## 3.2.3、可抢占式资源

3.2.4、不可抢占式资源

3.2.5、资源引发的死锁问题

3.3、死锁案例分析

3.4、死锁处理

3.4.1、预防死锁

3.4.2、避免死锁

3.4.3、检测死锁

3.4.4、解除死锁

@稀土掘金技术社区

一句话来概述死锁：**「死锁是指两个或两个以上的线程（或进程）在运行过程中，因为资源竞争而造成相互等待、相互僵持的现象」**，一般当程序中出现死锁问题后，若无外力介入，则不会解除“僵持”状态，它们之间会一直相互等待下去，直到天荒地老、海枯石烂~

“

当然，为了照顾一些不想看并发编程文章的小伙伴，这里也把之前的死锁栗子搬过来~

”

“

某一天竹子和熊猫在森林里捡到一把玩具弓箭，竹子和熊猫都想玩，原本说好一人玩一次的来，但是后面竹子耍赖，想再玩一次，所以就把弓一直拿在自己手上，而本应该轮到熊猫玩的，所以熊猫跑去捡起了竹子前面刚刚射出去的箭，然后跑回来之后便发生了如下状况：

熊猫道：竹子，快把你手里的弓给我，该轮到我玩了....

竹子说：不，你先把你手里的箭给我，我再玩一次就给你....

最终导致熊猫等着竹子的弓，竹子等着熊猫的箭，双方都不肯退步，结果陷入僵局场面....

”

比如上述这个栗子中，「竹子、熊猫」可以理解成两条线程，而「弓、箭」则可以理解成运行时所需的资源，由于双方各自占据对方所需的资源，因此就造就了死锁现象发生，此时想要解决这个问题，就必须第三者外力介入，把“违反约定”的竹子手中的弓拿过去给熊猫.....，然后等熊猫玩了之后，再给竹子，恢复之前原有的“执行顺序”。

```
SELECT * FROM `zz_account`;

+-----+-----+
| user_name | balance |
+-----+-----+
| 熊猫 | 6666666 |
| 竹子 | 8888888 |
+-----+-----+

-- T1事务: 竹子向熊猫转账
UPDATE `zz_account` SET balance = balance - 888 WHERE user_name = "竹子";
UPDATE `zz_account` SET balance = balance + 888 WHERE user_name = "熊猫";

-- T2事务: 熊猫向竹子转账
UPDATE `zz_account` SET balance = balance - 666 WHERE user_name = "熊猫";
UPDATE `zz_account` SET balance = balance + 666 WHERE user_name = "竹子";
```

上面有一张很简单的账户表，因为只是为了演示效果，所以其中仅设计了用户名和余额两个字段，紧接着有 T1、T2 两个事务，T1 中竹子向熊猫转账，而 T2 中则是熊猫向竹子转账，也就是一个相互转账的过程，此时来分析一下：

- ① T1 事务会先扣减竹子的账户余额，因此会修改数据，此时会默认加上排他锁。
- ② T2 事务也会先扣减熊猫的账户余额，因此同样会对熊猫这条数据加上排他锁。
- ③ T1 减完了竹子的余额后，准备获取锁把熊猫的余额加 888，但由于此时熊猫的锁被 T2 事务持有，T1 会陷入阻塞等待。
- ④ T2 减完熊猫的余额后，也准备获取锁把竹子的余额加 666，但此时竹子的锁被 T1 持有。

此时就会出现死锁问题，T1 等待 T2 释放锁、T2 等待 T1 释放锁，双方各自等待对方释放锁，一直如此僵持下去，最终就引发了死锁问题，那先来看看具体的 SQL 执行情况是什么样的呢？如下：

mysql> SELECT \* FROM zz\_account ;

user_name	balance
熊猫	6666666
竹子	8888888

2 rows in set (0.00 sec)

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE `zz\_account` SET balance = balance - 888 WHERE user\_name = "竹子";

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> UPDATE `zz\_account` SET balance = balance + 888 WHERE user\_name = "熊猫";

Query OK, 1 row affected (9.25 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql>

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE `zz\_account` SET balance = balance - 666 WHERE user\_name = "熊猫";

Query OK, 1 row affected (0.01 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> UPDATE `zz\_account` SET balance = balance + 666 WHERE user\_name = "竹子";

ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

mysql>

① 先查询一下账户表，其中存在两条数据！

② 开启一个事务：T1（执行竹子向熊猫转账的SQL）

③ 从竹子的余额中：-888

④ 对熊猫的余额：+888（这条SQL会阻塞，等待获取锁）

⑤ 此时当T2事务被MySQL介入回滚后，T2持有的锁会释放，此时T1事务获取锁成功并执行

⑥ 从熊猫的余额中：-666

⑦ 对竹子的余额：+666

⑧ MySQL检测到了死锁问题出现，然后回滚了当前事务T2

@稀土掘金技术社区

如上图所示，一步步的跟着标出的序号去看，最终会发现：当死锁问题出现时，MySQL 会自动检测并介入，强制回滚结束一个“死锁的参与者（事务）”，从而打破死锁的僵局，让另一个事务能继续执行。



看到这篇的小伙伴会问：为啥 MySQL 能自动检测死锁呀？其实这和死锁检测机制有关，后续再细说

但是安全一点，如果你也愿意自己做上这头险，那么千万别忘了在创建表后，基于 `user_name` 创建一个主键索引。

sql 复制代码

```
ALTER TABLE `zz_account` ADD PRIMARY KEY p_index(user_name);
```

如果你不为 `user_name` 字段加上主键索引，那是无法模拟出死锁问题的，这是为什么呢？还记得之前在《MySQL锁机制-记录锁》中聊到的一点嘛？在 `InnoDB` 中，如果一条 `SQL` 语句能命中索引执行，那就会加行锁，但如果无法命中索引加的就是表锁。

在上述给出的案例中，因为表中没有显示指定主键，同时也不存在一个唯一非空的索引，因此 `InnoDB` 会隐式定义一个 `row_id` 来维护聚簇索引的结构，但因为 `update` 语句中无法使用这个隐藏列，所以是走全表方式执行，因此就将整个表数据锁起来了。

而这里的四条 `update` 语句都是基于 `zz_account` 账户表在操作，因此两个事务竞争的是同一个锁资源，所以自然无法复现死锁现象，也就是 `T1` 修改时，`T2` 的第一条 `SQL` 也不能执行，会阻塞等待表锁的释放。

而当咱们显示的定义了主键索引后，`InnoDB` 会基于该主键字段去构建聚簇索引，因此后续的 `update` 语句可以命中索引，执行时自然获取的也是行级别的排他锁。

### 1.3、MySQL中死锁如何解决呢？

在之前关于死锁的并发文章中聊到过，对于解决死锁问题可以从多个维度出发，比如预防死锁、避免死锁、解除死锁等，而当死锁问题出现后该如何解决呢？一般只有两种方案：

- 锁超时机制：事务/线程在等待锁时，超出一定时间后自动放弃等待并返回。
- 外力介入打破僵局：第三者介入，将死锁情况中的某个事务/线程强制结束，让其他事务继续执行。

#### 1.3.1、MySQL的锁超时机制

在 `InnoDB` 中其实提供了锁的超时机制，也就是一个事务在长时间内无法获取到锁时，就会主动放弃等待，抛出相关的错误码及信息，然后返回给客户端。但这里的时间限制到底是多久呢？可以通过如下命令查询：

sql 复制代码

```
show variables like 'innodb_lock_wait_timeout';
```

Variable_name	Value
innodb_lock_wait_timeout	50

👍 60

💬 33

★ 收藏

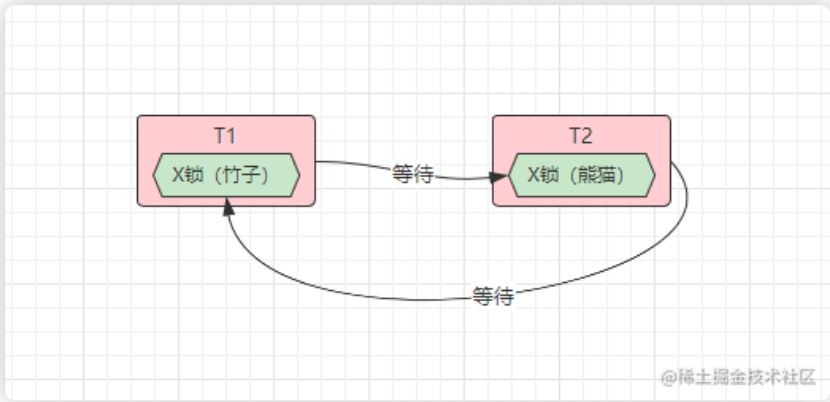
默认的锁超时时间是 `50s`，也就是在 `50s` 内未获取到锁的事务，会自动结束并返回。那也就意味着当死锁情况出现时，这个死锁过程最多持续 `50s`，然后其中就会有一个事务主动退出竞争，释放持有的锁资源，这似乎听起来蛮不错呀，但实际业务中，仅依赖超时机制去解除死锁是不够的，毕竟高并发情况下，`50s` 时间太长了，会导致越来越多的事务阻塞。

那么咱们能不能把这个参数调小一点呢？比如调到 `1s`，可以吗？当然可以，确实也能确保死锁发生后，在很短的时间内可以自动解除，但改掉了这个参数之后，也会影响正常事务等待锁的时间，也就是大部分未发生死锁，但需要等待锁资源的事务，在等待 `1s` 之后，就会立马报错并返回，这显然并不合理，毕竟容易误伤“友军”。

也正是由于依靠锁超时机制，略微有些不靠谱，因此 `InnoDB` 也专门针对于死锁问题，研发了一种检测算法，名为 `wait-for graph` 算法。

### 1.3.2、死锁检测算法 - wait-for graph

这种算法是专门用于检测死锁问题的，在该算法中会对于目前库中所有活跃的事务生成等待图，啥意思呢？以上述的死锁案例来看，在 `MySQL` 内部会生成一张这样的等待图：



也就是 `T1` 持有着「竹子」这条数据的锁，正在等待获取「熊猫」这条数据的锁，而 `T2` 事务持有「熊猫」这条数据的锁，正在等待获取「竹子」这条数据的锁，最终 `T1`、`T2` 两个事务之间就出现了等待闭环，因此当 `MySQL` 发现了这种等待闭环时，就会强制介入，回滚结束其中一个事务，强制打破该闭环，从而解除死锁问题。

但这个“等待图”只是为了方便理解画出来的，内部的实现其实存在些许差异，一起来聊一聊。

`wait-for graph` 算法被启用后，会要求 `MySQL` 收集两个信息：

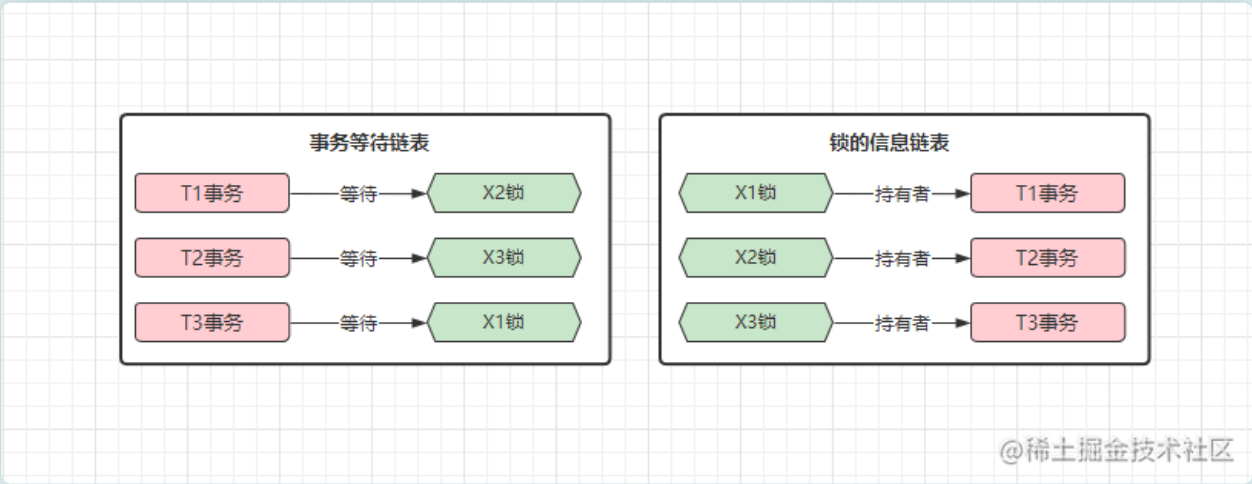
- 锁的信息链表：目前持有每个锁的事务是谁。

事务是否在等待获取其他锁，如果是，则再去看看另一个持有锁的事务，是否在等待其他锁.....，经过一系列的判断后，再看看是否会出现闭环，出现的话则介入破坏。



“

上面这个算法的过程，听起来似乎有些晕乎乎的，但实际上并不难，套个例子来理解，好比目前库中有 T1、T2、T3 三个事务、有 X1、X2、X3 三个锁，事务与锁的关系如下：



”

此时当 T3 事务需要阻塞等待获取 X1 锁时，就会触发一次 wait-for graph 算法，流程如下：

- ①先根据 T3 要获取的 X1 锁，在「锁的信息链表」中找到 X1 锁的持有者 T1。
- ②再在「事务等待链表」中查找，看看 T1 是否在等待获取其他锁，此时会得知 T1 等待 X2。
- ③再去「锁的信息链表」中找到 X2 锁的持有者 T2，再看看 T2 是否在阻塞等待获取其他锁。
- ④再在「事务等待链表」中查找 T2，发现 T2 正在等待获取 X3 锁，再找 X3 锁的持有者。

经过上述一系列算法过程后，最终会发现 X3 锁的持有者为 T3，而本次算法又正是 T3 事务触发的，此时又回到了 T3 事务，也就代表着产生了“闭环”，因此也可以证明这里出现了死锁现象，所以 MySQL 会强制回滚其中的一个事务，来抵达解除死锁的目的。

“

但出现死锁问题时，MySQL 会选择哪个事务回滚呢？之前分析过，当一个事务在执行 SQL 更改数据时，都会记录在 Undo-log 日志中，Undo 量越小的事务，代表它对数据的更改越少，同时回滚的代价最低，因此会选择 Undo 量最小的事务回滚（如若两个事务的 Undo 量相同，会选择回滚触发死锁的事务）。

”

同时，可以通过 innodb\_deadlock\_detect=on|off 这个参数，来控制是否开启死锁检测机制。

“

死锁检测机制在 MySQL 后续的高版本中是默认开启的，但实际上死锁检测的开销不小，上面三个并发事务阻塞时，会对「事务等待链表、锁的信息链表」共计检索六次，那当阻塞的并发事务越来越多时，检测的效率也会呈



### 1.3.3、如何避免死锁产生？



因为死锁的检测过程较为耗时，所以尽量不要等死锁出现后再去解除，而是尽量调整业务避免死锁的产生，一般来说可以从如下方面考虑：

- 合理的设计索引结构，使业务 SQL 在执行时能通过索引定位到具体的几行数据，减小锁的粒度。
- 业务允许的情况下，也可以将隔离级别调低，因为级别越低，锁的限制会越小。
- 调整业务 SQL 的逻辑顺序，较大、耗时较长的事务尽量放在特定时间去执行（如凌晨对账...）。
- 尽可能的拆分业务的粒度，一个业务组成的大事务，尽量拆成多个小事务，缩短一个事务持有锁的时间。
- 如果没有强制性要求，就尽量不要手动在事务中获取排他锁，否则会造成一些不必要的锁出现，增大产生死锁的几率。
- .....

其实简单来说，也就是在业务允许的情况下，尽量缩短一个事务持有锁的时间、减小锁的粒度以及锁的数量。

“

同时也要记住：当 MySQL 运行过程中产生了死锁问题，那这个死锁问题以后绝对会再次出现，当死锁被 MySQL 自己解除后，一定要记住去排除业务 SQL 的执行逻辑，找到产生死锁的业务，然后调整业务 SQL 的执行顺序，这样才能从根源上避免死锁产生。

”

## 二、锁机制的底层实现原理

对于 MySQL 的锁机制究竟是如何实现的呢？对于这点其实很少有资料去讲到，一般都是停留在锁机制的表层阐述，比如锁粒度、锁类型的划分，但既然咱们讲了锁机制，那也就顺便聊一下它的底层实现。

### 2.1、锁的内存结构

在 Java 中，Synchronized 锁是基于 Monitor 实现的，而 ReentrantLock 又是基于 AQS 实现的，那 MySQL 的锁是基于啥实现的呢？想要搞清楚这点，得先弄明白锁的内存结构，先看图：

InnoDB 引擎中，每个锁对象在内存中的结构如上，其中记录的信息也比较多，先全部理清楚后再聊聊锁的实现。

#### 2.1.1、锁的事务信息

其中记录着当前的锁结构是由哪个事务生成的，记录的是指针，指向一个具体的事务。



60



33



收藏



这个是行锁的特有信息，对于行锁来说，需要记录一下加锁的行数据  哪个索引、哪个节点，记录的也是指针。

2.1.3、锁粒度信息

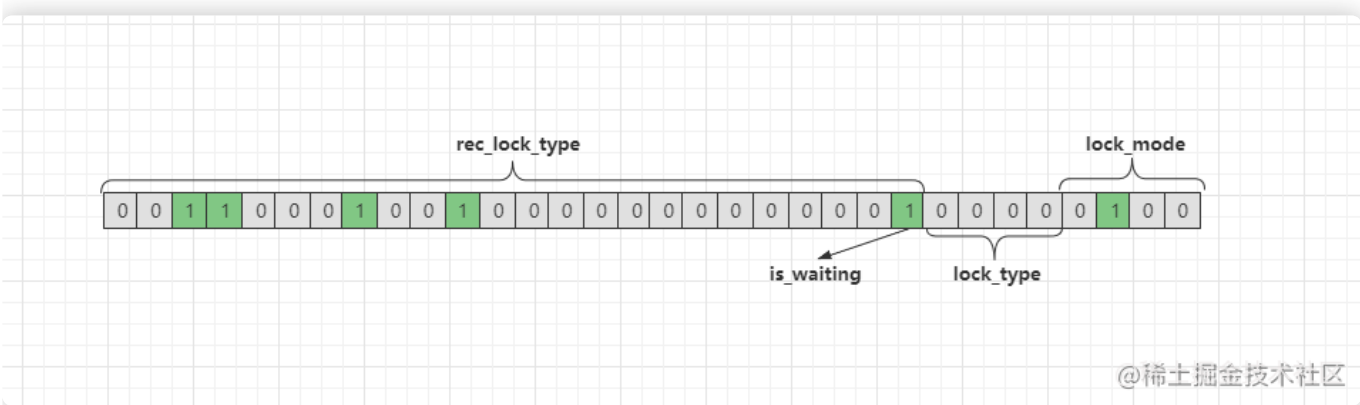
这个略微有些复杂，对于不同粒度的锁，其中存储的信息也并不同，如果是表锁，其中就记录了一下是对哪张表加的锁，以及表的一些其他信息。

但如果锁粒度是行锁，其中记录的信息更多，有三个较为重要的：

- `Space ID`：加锁的行数据，所在的表空间 `ID`。
- `Page Number`：加锁的行数据，所在的页号。
- `n_bits`：使用的比特位，对于一页数据中，加了多少个锁（后续结合讲）。

2.1.4、锁类型信息

对于锁结构的类型，在内部实现了复用，采用一个 `32bit` 的 `type_mode` 来表示，这个 `32bit` 的值可以拆为 `lock_mode`、`lock_type`、`rec_lock_type` 三部分，如下：



- `lock_mode`：表示锁的模式，使用低四位。
  - `0000/0`：表示当前锁结构是共享意向锁，即 `IS` 锁。
  - `0001/1`：表示当前锁结构是排他意向锁，即 `IX` 锁。
  - `0010/2`：表示当前锁结构是共享锁，即 `S` 锁。
  - `0011/3`：表示当前锁结构是排他锁，即 `X` 锁。
  - `0100/4`：表示当前锁结构是自增锁，即 `AUTO-INC` 锁。
- `lock_type`：表示锁的类型，使用低位中的 `5~8` 位。
  - `LOCK_TABLE`：当第 `5` 个比特位是 `1` 时，表示目前是表级锁。
  - `LOCK_REC`：当第 `6` 个比特位是 `1` 时，表示目前是行级锁。
- `rec_lock_type`：表示行锁的具体类型，使用其余位。
  - `LOCK_ORDINARY`：当高 `23` 位全零时，表示目前是临键锁。
  - `LOCK_GAP`：当第 `10` 位是 `1` 时，表示目前是间隙锁。





## 2.2、InnoDB的锁实现



上面已经分析了 MySQL 的锁对象结构，接着来设想一个问题：

“

如果一个事务同时需要对表中的 1000 条数据加锁，会生成 1000 个锁结构吗？

”

如果这里是 SQL Server 数据库，那绝对会生成 1000 个锁结构，因为它的行锁是加在行记录上的，但 MySQL 锁机制并不相同，因为 MySQL 是基于事务实现的锁，啥意思呢？来看看：

- ①目前对表中不同行记录加锁的事务是同一个。
- ②需要加锁的记录在同一个页面中。
- ③目前事务加锁的类型都是相同的。
- ④锁的等待状态也是相同的。

当上述四点条件被满足时，符合条件的行记录会被放入到同一个锁结构中，好比以上面的问题为例：

“

假设加锁的 1000 条数据分布在 3 个页面中，同时表中没有其他事务在操作，加的都是同一类型的锁。

”

此时依据上述的前提条件，那在内存中仅会生成三个锁结构，能够很大程度上减少锁结构的数量。总之情况再复杂，也不会像 SQL Server 般生成 1000 个锁对象，那样开销太大了！

## 2.3、MySQL获取锁的过程

当一个事务需要获取某个行锁时，首先会看一下内存中是否存在这条数据的锁结构，如果存在则生成一个锁结构，将其 `is_waiting` 对应的比特位改为 1，表示目前事务在阻塞等待获取该锁，当其他事务释放锁后，会唤醒当前阻塞的事务，然后会将其 `is_waiting` 改为 0，接着执行 SQL。

实际上会发现这个过程并不复杂，唯一有些难理解的点就在于：事务获取锁时，是如何在内存中，判断是否已经存在相同记录的锁结构呢？还记得锁结构中会记录的一个信息嘛？也就是「锁粒度信息」，如果是表锁，会记录表信息，如果是行锁，会记录表空间、页号等信息。在事务获取锁时，就是去看内存中，已存在的锁结构的这个信息，来判断是否存在其他事务获取了锁。

“

拿表锁来说，当事务要获取一张表的锁时，就会根据表名看一下其他锁结构，有没有获取当前这张表的锁，如果已经获取，看一下已经存在的表锁和目前要加的表锁，是否会存在冲突，冲突的话 `is_waiting=1`，反之 `is_waiting=0`，而行锁也是差不多的过程。

”

其实看起来之后大家会发现，MySQL 的锁机制实现，与常规的锁实现有些不一样，一般的锁机制都是基于持有标识+等待队列实现的，而 MySQL 则是略微有些不一样。



### 三、事务隔离机制的底层实现

对于事务隔离机制的底层实现，其实在前面的章节中简单聊到过，对于并发事务造成的各类问题，在不同的隔离级别实际上，是通过不同粒度、类型的锁以及 MVCC 机制来解决的，也就是调整了并发事务的执行顺序，从而避免了这些问题产生，具体是如何做的呢？先来看看 DBMS 中对各隔离级别的要求。

- RU /读未提交级别：要求该隔离级别下解决脏写问题。
- RC /读已提交级别：要求该隔离级别下解决脏读问题。
- RR /可重复读级别：要求该隔离级别下解决不可重复读问题。
- Serializable /序列化级别：要求在该隔离级别下解决幻读问题。

虽然 DBMS 中要求在序列化级别再解决幻读问题，但在 MySQL 中，RR 级别中就已经解决了幻读问题，因此 MySQL 中可以将 RR 级别视为最高级别，而 Serializable 级别几乎用不到，因为序列化级别中解决的问题，在 RR 级别中基本上已经解决了，再将 MySQL 调到 Serializable 级别反而会降低性能。

当然，RR 级别下有些极端的情况，依旧会出现幻读问题，但线上 100% 不会出现，这点后续聊，先来看看各大隔离级别在 MySQL 中是如何实现的。

#### 3.1、RU (Read Uncommitted) 读未提交级别的实现

对于 RU 级别而言，从它名字上就可以看出来，该隔离级别下，一个事务可以读到其他事务未提交的数据，但同时要求解决脏写（更新覆盖）问题，那思考一下该怎么满足这个需求呢？先来看看不加锁的情况：

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

----- 请按照标出的序号阅读代码!!! -----

```
-- ①开启一个事务T1
begin;

-- ③修改 ID=1 的姓名为 竹子
UPDATE `zz_users` SET user_name = "竹子" WHERE user_id = 1;

-- ⑥提交T1
commit;
```

👍 60

💬 33

★ 收藏

```
begin;
```

— ④这里可以读取到T1中还未提交的 竹子 记录

```
SELECT * FROM `zz_users` WHERE user_id = 1;
```



— ⑤T2中再次修改姓名为 黑熊

```
UPDATE `zz_users` SET user_name = "黑熊" WHERE user_id = 1;
```

— ⑦提交T2

```
commit;
```

假设上述两个事务并发执行时，都不加锁，T2 自然可以读取到 T1 修改后但还未提交的数据，但当 T2 再次修改 ID=1 的数据后，两个事务一起提交，此时就会出现 T2 覆盖 T1 的问题，这也就是脏写问题，而这个问题是不允许存在的，所以需要解决，咋解决呢？

“

写操作加排他锁，读操作不加锁！

”

还是上述的例子，当写操作加上排他锁后，T1 在修改数据时，当 T2 再次尝试修改相同的数据，也要获取排他锁，因此 T1、T2 两个事务的写操作会相互排斥，T2 就需要阻塞等待。但因为读操作不会加锁，因此当 T2 尝试读取这条数据时，自然可以读到数据。

“

来分析一下，因为写-写会排斥，但写-读不会排斥，因此也满足了 RU 级别的要求，即可以读到未提交的数据，但是不允许出现脏写问题。

”

最终经过这一系列的讲解后，能够得知 MySQL-RU 级别的实现原理，「即写操作加排他锁，读操作不加锁！」

### 🔗 3.2、RC (Read Committed) 读已提交级别的实现 ➡

理解了 RU 级别的实现后，再看看 RC，RC 级别要求解决脏读问题，也就是一个事务中，不允许读另一个事务还未提交的数据，咋实现呢？

“

写操作加排他锁，读操作加共享锁！

”

这样一想，似乎好像没问题，还是以之前的例子来说，因为 T1 在修改数据，所以会对 ID=1 的数据加上排他锁，此时 T2 想要获取共享锁读数据时，T1 的排他锁就会排斥 T2，因此 T2 需要等到 T1 事务结束后才能读数据。



首页 ▾



因为 T2 需要等待 T1 结束后才能读，既然 T1 都结束了，那也就代表着 T1 事务要么回滚了，T2 读上一个事务提交的数据；要么 T1 提交了，T2 读 T1 提交的数据，总之 T2 读到的数据绝对是提交过的数据。



这种方式的确能解决脏读问题，但似乎也会将所有并发事务串行化，会导致 MySQL 整体性能下降，因此 MySQL 引入了一种技术，也就是上篇聊到的《MVCC机制》，在每次 select 查询数据时，都会生成一个 ReadView 快照，然后依据这个快照去选择一个可读的数据版本。



因此对于 RC 级别的底层实现，对于写操作会加排他锁，而读操作会使用 MVCC 机制。



但由于每次 select 时都会生成 ReadView 快照，此时就会出现下述问题：

sql 复制代码

— ①T1事务中先读取一次 ID=1 的数据

```
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

— ②T2事务中修改 ID=1 的姓名为 竹子 并提交

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

— ③T1事务中再读取一次 ID=1 的数据

```
SELECT * FROM `zz_users` WHERE user_id = 1;
```

user_id	user_name	user_sex	password	register_time
1	竹子	女	6666	2022-08-14 15:22:01

此时观察这个案例，明明是在一个事务中查询同一条数据，结果两次查询的结果并不一致，这也是所谓的不可重复读的问题。

### 3.3、RR (Repeatable Read) 可重复读级别的实现



在 RC 级别中，虽然解决了脏读问题，但依旧存在不可重复读问题，而 RR 级别中，就是要确保一个事务中的多次读取结果一致，即解决不可重复读问题，咋解决呢？两种方案：

- ①查询时，对目标数据加上临键锁，即读操作执行时，不允许其他事务改动数据。
- ② MVCC 机制的优化版：一个事务中只生成一次 ReadView 快照。

相较于第一种方案，第二种方案显然性能会更好，因为第一种方案不允许读-写、写-读事务共存，而第二种方案则支持读



60



33



收藏

写操作加排他锁，对读操作依旧采用 MVCC 机制，但 RR 级别中，一个事务中只有首次 select 会生成 ReadView 快照。



sql 复制代码

— ①T1事务中先读取一次 ID=1 的数据

```
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

— ②T2事务中修改 ID=1 的姓名为 竹子 并提交

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

— ③T1事务中再读取一次 ID=1 的数据

```
SELECT * FROM `zz_users` WHERE user_id = 1;
```

user_id	user_name	user_sex	password	register_time
1	竹子	女	6666	2022-08-14 15:22:01

还是以这个场景为例，在 RC 级别中，会对于 T1 事务的每次 SELECT 都生成快照，因此当 T1 第二次查询时，生成的快照中就能看到 T2 修改后提交的数据。但在 RR 级别中，只有首次 SELECT 会生成快照，当第二次 SELECT 操作出现时，依旧会基于第一次生成的快照查询，所以就能确保同一个事务中，每次看到的数据都是相同的。



也正是由于 RR 级别中，一个事务仅有首次 select 会生成快照，所以不仅仅解决了不可重复读问题，还解决了幻读问题，举个例子：



sql 复制代码

— 先查询一次用户表，看看整张表的数据

```
SELECT * FROM `zz_users`;
```

user_id	user_name	user_sex	password	register_time
1	熊猫	女	6666	2022-08-14 15:22:01
2	竹子	男	1234	2022-09-14 16:17:44
3	子竹	男	4321	2022-09-16 07:42:21
4	猫熊	女	8888	2022-09-27 17:22:59
9	黑竹	男	9999	2022-09-28 22:31:44

— ①T1事务中，先查询所有 ID>=4 的用户信息

```
SELECT * FROM `zz_users` WHERE user_id >= 4;
```



```

+-----+-----+-----+-----+-----+
| user_id | user_name | user_sex | password | register_time |
+-----+-----+-----+-----+-----+
| 4 | 猫熊 | 女 | 1111 | 2022-09-27 17:22:59 |
| 6 | 棕熊 | 男 | 7777 | 2022-10-02 16:21:33 |
| 9 | 黑竹 | 男 | 1111 | 2022-09-28 22:31:44 |
+-----+-----+-----+-----+-----+

-- ②T1事务中，再将所有 ID>=4 的用户密码重置为 1111
UPDATE `zz_users` SET password = "1111" WHERE user_id >= 4;

-- ③T2事务中，插入一条 ID=6 的用户数据
INSERT INTO `zz_users` VALUES (6,"棕熊","男","7777","2022-10-02 16:21:33");

-- ④提交事务T2
commit;

-- ⑤T1事务中，再次查询所有 ID>=4 的用户信息
+-----+-----+-----+-----+-----+
| user_id | user_name | user_sex | password | register_time |
+-----+-----+-----+-----+-----+
| 4 | 猫熊 | 女 | 1111 | 2022-09-27 17:22:59 |
| 6 | 棕熊 | 男 | 7777 | 2022-10-02 16:21:33 |
| 9 | 黑竹 | 男 | 1111 | 2022-09-28 22:31:44 |
+-----+-----+-----+-----+-----+

```

此时会发现，明明 T1 中已经将所有 ID>=4 的用户密码重置为 1111 了，结果改完再次查询会发现，表中依旧存在一条 ID>=4 的数据：棕熊，而且密码未被重置，这似乎产生了幻觉一样。


“

如果是 RC 级别，因为每次 select 都会生成快照，因此会出现这个幻读问题，但 RR 级别中因为只有首次查询会生成 ReadView 快照，因此上述案例放在 RR 级别的 MySQL 中，T1 看不到 T2 新增的数据，因此 MySQL-RR 级别也解决了幻读问题。

”

小争议：MVCC机制是否彻底解决了幻读问题呢？

先上定论，MVCC 并没有彻底解决幻读问题，在一种奇葩的情况下依旧会出现问题，先来看例子：


 sql 复制代码

```

-- 开启一个事务T1
begin;
-- 查询表中 ID>10 的数据
SELECT * FROM `zz_users` where user_id > 10;
Empty set (0.01 sec)

```

因为用户表中不存在 ID>10 的数据，所以 T1 查询时没有结果，再继续往下看。

 sql 复制代码

```

-- 再开启一个事务T2
begin;
-- 向表中插入一条 ID=11 的数据
INSERT INTO `zz_users` VALUES (11,"墨竹","男","2222","2022-10-07 23:24:36");
-- 提交事务T2

```



```
— 在T1事务中，再次查询表中 ID>10 的数据
SELECT * FROM `zz_users` where user_id > 10;
Empty set (0.01 sec)
```



结果很明显，依旧未查询到 ID>10 的数据，因为这里是通过第一次生成的快照文件在读，所以读不到 T2 新增的“幻影数据”，似乎没问题对嘛？接着往下看：

```
— 在T1事务中，对 ID=11 的数据进行修改
UPDATE `zz_users` SET `password` = "1111" where `user_id` = 11;
```

```
— 在T1事务中，再次查询表中 ID>10 的数据
SELECT * FROM `zz_users` where user_id > 10;
```

user_id	user_name	user_sex	password	register_time
11	墨竹	男	1111	2022-10-07 23:24:36

嗯？！？？此时会发现，T1 事务中又能查询到 ID=11 的这条幻影记录了，这是啥原因导致的呢？因为我们在 T1 中修改了 ID=11 的数据，在《MVCC机制原理剖析》中曾讲过 MVCC 通过快照检索数据的过程，这里 T1 根据原本的快照文件检索数据时，因为发现 ID=11 这条数据上的隐藏列 trx\_id 是自己，因此就能看到这条幻影数据了。

“

实际上这个问题有点四不像，可以理解成幻读问题，也可以理解成是不可重复读问题，总之不管怎么说，就是 MVCC 机制存在些许问题！但这种情况线下一般不会发生，毕竟不同事务之间都是互不相知的，在一个事务中，不可能去主动修改一条“不存在”的记录。

”

但如若你实在不放心，想要彻底杜绝任何风险的出现，那就直接将事务隔离级别调整到 Serializable 即可。

### 3.4、Serializable序列化级别的实现

前面已经将 RU、RC、RR 三个级别的实现原理弄懂了，最后来看看最高的 Serializable 级别，在这个级别中，要求解决所有可能会因并发事务引发的问题，那怎么做呢？比较简单：

“

所有写操作加临键锁（具备互斥特性），所有读操作加共享锁。

”

由于所有写操作在执行时，都会获取临键锁，所以写-写、读-写、写-读这类并发场景都会互斥，而由于读操作加的是共享锁，因此在 Serializable 级别中，只有读-读场景可以并发执行。

在本章中，实则更多的是对《MySQL事务篇》、《MySQL锁机制》、《MySQL-MVCC机制》的补充以及汇总，在本篇中补齐了MySQL死锁分析、锁实现原理、事务隔离机制原理等内容，也结合事务、锁、MVCC机制三者的知识点，彻底理清了MySQL不同隔离级别下的实现，最后做个简单的小总结：

- RU级别：读操作不加锁，写操作加排他锁。
- RC级别：读操作使用MVCC机制，每次SELECT生成快照，写操作加排他锁。
- RR级别：读操作使用MVCC机制，首次SELECT生成快照，写操作加临键锁。
- 序列化级别：读操作加共享锁，写操作加临键锁。

级别/场景	读-读	读-写/写-读	写-写
RU级别	并行执行	并行执行	串行执行
RC级别	并行执行	并行执行	串行执行
RR级别	并行执行	并行执行	串行执行
序列化级别	并行执行	串行执行	串行执行



到这里，MySQL事务机制、锁机制、MVCC机制、隔离机制就彻底剖析完毕啦~



分类：后端

标签：MySQL 数据库 Java

文章被收录于专栏：



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、事务与锁机制、存储引擎剖析...

关注专栏

相关课程

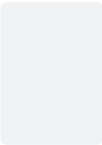


VIP 深入理解 TCP 协议：从原理到实战

挖坑的张师傅 LV.5

¥49.9

7350购买



图解 Kafka 之实战指南

朱小厮 LV.3

¥29.9

4411购买



输入评论 (Enter换行, Ctrl + Enter发送)



全部评论 33

🔴 最新


🔥 最热

 captain\_k  搬砖工 @ 无 7天前

竹老师您好，关于锁的获取过程有些不懂。锁是存在内存中的吗，要获取一个行锁怎么知道这个行记录有多少锁，怎么根据行记录找到对应的锁？辅助索引的行级锁是锁定到主键索引文件上的吗，辅助索引的间隙锁呢？

👍 点赞

💬 6

 竹子爱熊猫 (作者) 7天前

非聚簇索引会先判断辅助索引键上有没有锁，然后再去判断一下主键索引有没有锁，如果都没有就会开始加锁的动作。

同时你想要搞明白获取锁的流程，你得先理解明白内存中的锁结构，里面记录着锁的信息，每个事务生成的锁对象中，都会记录着锁对应的表空间、数据段、数据页、行数的信息，当尝试加锁时，会通过要加锁的数据位置，去内存中看一下有没有存在对应的锁对象，如果有代表已经有线程持有锁了，则去看一下锁的比特位，当前要获取的锁范围有没有加锁，有的话当前事务生成的锁对象则会陷入阻塞状态。...

展开

👍 点赞

💬 回复

 captain\_k 回复 竹子爱熊猫 6天前

juc的加锁是对对象加锁，比如一个ReentrantLock类型的lock对象，在内存中是唯一的，多线程中可以拿到这个变量进行判断。事务对行数据加锁，会在内存中生成一个锁对象（锁结构），锁结构中记录着行数据，但是行数据上没有逆向关联锁对象（有吗？有的话我大概理解了，关联的话是把行数据加载到内存中关联的吗？），当另一个事务要对该行数据加锁时是怎么判断有无其他锁的，不是生成一个新的锁对象了吗？也就是2.3章节“当一个事务需要获取某个行锁时，首先会看一下内存中是否存在这条数据的锁结构”这句话不理解。我的思路有点陷入牛角尖了，还请老师指点下

“非聚簇索引会先判断辅助索引键上有没有锁，然后再去判断一下主键索引有没有锁，如果都没有就会开始加锁...”

👍 点赞

💬 回复


查看更多回复

 欠睡觉的猫  21天前

请教一下啊，锁粒度信息的Space ID和Page Number是指什么呢，表空间和页号感觉好陌生的名词😭

👍 点赞

💬 3

 竹子爱熊猫 (作者) 21天前

你如果对这块比较陌生，那应该是对MySQL底层这块还没摸明白，MySQL的表会有表空间，所谓的Space ID也就是指当前这个锁是基于哪个表空间上锁的，Page Number则是表空间中具体的数据位置，因为所有的数据都会按数据页为单位来存储。

拿日常生活来举例子，一张表就类似于一个大的工厂，一个工厂会分为不同的片区，而所谓的表空间则是这些片区，一个片区中会存在多条流水线，一条流水线上有一个个的工人，这里的流水线就是数据页，里面的工人就是一条条数据...

展开

“你如果对这块比较陌生，那应该是对MySQL底层这块还没摸明白，MySQL的表会有表空间，所谓的Space ID...”

点赞

回复



查看更多回复



xiaobinqt LV.3

26天前

请教一个问题，这里的「Undo量」是指 undo-log 中的某条记录的版本链表的长度吗？



点赞

3 回复



竹子爱熊猫（作者）

25天前

不是的，是指一个事务写成功的数据量，比如：  
T1事务：新增两条数据，删除一条数据，修改三条数据。  
T2事务：修改两条数据。

假设T1已经成功新增了两条数据，删除了一条数据，修改了两条数据，T2修改了一条数据，此时T1事务的Undo量就可以理解成是5，而T2的Undo量则可以理解成是1，所以两个事务发生死锁时会回滚T2。...

展开

点赞

回复



xiaobinqt 回复 竹子爱熊猫

22天前

我这种小白很多东西不是很理解，感谢作者这么详细的回答，👍

“不是的，是指一个事务写成功的数据量，比如：...”

点赞

回复

查看更多回复



忙碌的尘埃 LV.2 JY.3

1月前

写的好👍

点赞

1 回复



竹子爱熊猫（作者）

1月前

三克油~，感谢认可

点赞

回复



宁在春 LV.5 JY.6 @ Java

1月前

竹哥，好奇一下下，如何才会学到这样的知识~👍

点赞

3 回复

 首页 ▾

捞

Q





尾篇的时候，我会放一下我参考过的书籍、文章资料的，看了近十本书，认真阅读了几百篇其他博主的文章，点进过的文章就具体记不清了😂，一个完整性的输出，自己肯定要能坚持的住，才能啃的下来。

（其实这也是我写连续文章的原因，我目前更新的文章中，9以上都是连载性质的，因为从我自己的经验来说，无论是书籍也好，还是其他博主的文章，多多少少都会有些缺失，比如书籍中有些细节不会提，而有些文章只有单独几个...

展开

👍 3

💬 回复

 captain\_k

18天前

mysql技术内幕 innodb 这本书里关于这块写的也挺详细的，好像在事务那篇，可以看完博主的文章再去阅读

👍 1

💬 回复

查看更多回复 ▾



宁在春

LV.5

JY.6



@ Java

1月前

又来竹哥这逛了，这篇文章都让我放了两天才看~ 看完仍然是收获颇多~ 给我竹哥三连！！！哈哈

👍 点赞

💬 1

 竹子爱熊猫 （作者）

1月前

三克油~，这篇是前面《事务篇》、《锁机制》、《MVCC机制》的整合篇。

👍 点赞

💬 回复

 StoneDB

StoneDB

LV.2

JY.5

数据库架构师 @ 石原...

1月前

持续追更

👍 点赞

💬 1

 竹子爱熊猫 （作者）

1月前

欢迎大佬光临~

👍 点赞

💬 回复



那个学java的上...

JY.3

1月前

老哥下一个系列是关于什么的啊

👍 点赞

💬 1

 竹子爱熊猫 （作者）

1月前

写完MySQL之后，先回去把《网络编程》更完，然后其他的就从已建的专栏里面挑一个写。

👍 点赞

💬 回复



用户526251261...

JY.3

1月前



👍 点赞

💬 1

👍 60

💬 33

☆ 收藏



1月前

👍 1    💬 1

1月前

👍 点赞    💬 回复

1月前

点赞 1

1月前

👍 点赞    💬 回复

☆ 收藏



首页 ▾

捞



## 2022，前端的天🌩要怎么变？

👁 17.6w    👍 1698    💬 733



竹子爱熊猫 | 2月前 | 数据库 · MySQL · Java

## (五)MySQL索引应用篇：建立索引的正确姿势与使用索引的最佳指南！

👁 5625    👍 85    💬 58

竹子爱熊猫 | 2月前 | Java · 数据库 · MySQL

## (二)全解MySQL：一条SQL语句从诞生至结束的多姿多彩历程！

👁 4099    👍 88    💬 38

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    👍 10434    💬 356

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    👍 4635    💬 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    👍 3504    💬 314

神三元 | 2年前 | JavaScript · 面试

## (建议精读) HTTP灵魂一问，巩固你的 HTTP 知识体系

👁 21.7w    👍 4707    💬 178

大海我来了 | 1年前 | JavaScript

## 死磕 36 个 JS 手写题（搞懂后，提升真的大）

👁 13.8w    👍 3943    💬 221

竹子爱熊猫 | 1月前 | 数据库 · MySQL · Java

## (七)MySQL事务篇：ACID原则、事务隔离级别及事务机制原理剖析

👁 2701    👍 57    💬 35

前端鲨鱼哥 | 1年前 | JavaScript

## 最全的 Vue 面试题+详解答案

👁 20.3w    👍 3929    💬 217

~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



60



33



收藏



李小白白 | 3年前 | MySQL

记一次Mysql并发"死锁"，引出的问题及讨论



5926 11 4

wooyoo | 5年前 | MySQL

记录一次 Mysql 死锁排查过程

9411 66 3

神三元 | 2年前 | JavaScript · ECMAScript 6

(1.6w字)浏览器灵魂之问，请问你能接得住几个？

19.0w 3867 267

友情链接:

娱乐: 赤伶惊天下, 表白杨老板 沙雕的我在乙游修罗场里反复横跳 被休后, 王爷想吃回头草 万古至尊 centos7关闭x server后黑屏 window 杀掉java进程命令 python对图片处理函数 java 读取p12 spark structured streaming python example 实体清空 java