

讲堂 > MySQL实战45讲 > 文章详情

## 20 | 幻读是什么，幻读有什么问题？

2018-12-28 林晓斌




### 20 | 幻读是什么，幻读有什么问题？

朗读人：林晓斌 19'24" | 17.77M

在上一篇文章最后，我给你留了一个关于加锁规则的问题。今天，我们就从这个问题说起吧。

为了便于说明问题，这一篇文章，我们就先使用一个小一点儿的表。建表和初始化语句如下（为了便于本期的例子说明，我把上篇文章中用到的表结构做了点儿修改）：

```
1 CREATE TABLE `t` (  
2   `id` int(11) NOT NULL,  
3   `c` int(11) DEFAULT NULL,  
4   `d` int(11) DEFAULT NULL,  
5   PRIMARY KEY (`id`),  
6   KEY `c` (`c`)  
7 ) ENGINE=InnoDB;  
8  
9 insert into t values(0,0,0),(5,5,5),  
10 (10,10,10),(15,15,15),(20,20,20),(25,25,25);
```

 复制代码

这个表除了主键 id 外，还有一个索引 c，初始化语句在表中插入了 6 行数据。

上期我留给你的问题是，下面的语句序列，是怎么加锁的，加的锁又是什么时候释放的呢？

```
1 begin;
2 select * from t where d=5 for update;
3 commit;
```

[复制代码](#)

比较好理解的是，这个语句会命中  $d=5$  的这一行，对应的主键  $id=5$ ，因此在 `select` 语句执行完成后， $id=5$  这一行会加一个写锁，而且由于两阶段锁协议，这个写锁会在执行 `commit` 语句的时候释放。

由于字段  $d$  上没有索引，因此这条查询语句会做全表扫描。那么，其他被扫描到的，但是不满足条件的 5 行记录上，会不会被加锁呢？

我们知道，InnoDB 的默认事务隔离级别是可重复读，所以本文接下来没有特殊说明的部分，都是设定在可重复读隔离级别下。

## 幻读是什么？

现在，我们就来分析一下，如果只在  $id=5$  这一行加锁，而其他行的不加锁的话，会怎么样。

下面先来看一下这个场景：

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ result: (5,5,5)		
T2		update t set d=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/ result: (0,0,0),(5,5,5)		
T4			insert into t values(1,1,5);
T5	select * from t where d=5 for update; /*Q3*/ result: (0,0,0),(1,1,1),(5,5,5)		
T6	commit;		

图 1 假设只在  $id=5$  这一行加行锁

可以看到，session A 里执行了三次查询，分别是 Q1、Q2 和 Q3。它们的 SQL 语句相同，都是 `select * from t where d=5 for update`。这个语句的意思你应该很清楚了，查所有  $d=5$  的

行，而且使用的是当前读，并且加上写锁。现在，我们来看一下这三条 SQL 语句，分别会返回什么结果。

1. Q1 只返回 id=5 这一行；
2. 在 T2 时刻，session B 把 id=0 这一行的 d 值改成了 5，因此 T3 时刻 Q2 查出来的是 id=0 和 id=5 这两行；
3. 在 T4 时刻，session C 又插入一行 (1,1,5)，因此 T5 时刻 Q3 查出来的是 id=0、id=1 和 id=5 的这三行。

其中，Q3 读到 id=1 这一行的现象，被称为“幻读”。也就是说，幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。

这里，我需要对“幻读”做一个说明：

1. 在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的。因此，幻读在“当前读”下才会出现。
2. 上面 session B 的修改结果，被 session A 之后的 select 语句用“当前读”看到，不能称为幻读。幻读仅专指“新插入的行”。

如果只从第 8 篇文章[《事务到底是隔离的还是不隔离的？》](#)我们学到的事务可见性规则来分析的话，上面这三条 SQL 语句的返回结果都没有问题。

因为这三个查询都是加了 for update，都是当前读。而当前读的规则，就是要能读到所有已经提交的记录的最新值。并且，session B 和 session C 的两条语句，执行后就会提交，所以 Q2 和 Q3 就是应该看到这两个事务的操作效果，而且也看到了，这跟事务的可见性规则并不矛盾。

但是，这是不是真的没问题呢？

不，这里还真就有问题。

## 幻读有什么问题？

**首先是语义上的。**session A 在 T1 时刻就声明了，“我要把所有 d=5 的行锁住，不准别的事务进行读写操作”。而实际上，这个语义被破坏了。

如果现在这样看感觉还不明显的话，我再往 session B 和 session C 里面分别加一条 SQL 语句，你再看看会出现什么现象。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/		
T2		update t set d=5 where id=0; update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		
T4			insert into t values(1,1,5); update t set c=5 where id=1;
T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 2 假设只在 id=5 这一行加行锁 -- 语义被破坏

session B 的第二条语句 `update t set c=5 where id=0`，语义是“我把 id=0、d=5 这一行的 c 值，改成了 5”。

由于在 T1 时刻，session A 还只是给 id=5 这一行加了行锁，并没有给 id=0 这行加上锁。因此，session B 在 T2 时刻，是可以执行这两条 update 语句的。这样，就破坏了 session A 里 Q1 语句要锁住所有 d=5 的行的加锁声明。

session C 也是一样的道理，对 id=1 这一行的修改，也是破坏了 Q1 的加锁声明。

### 其次，是数据一致性的问题。

我们知道，锁的设计是为了保证数据的一致性。而这个一致性，不止是数据库内部数据状态在此刻的一致性，还包含了数据和日志在逻辑上的一致性。

为了说明这个问题，我给 session A 在 T1 时刻再加一个更新语句，即：`update t set d=100 where d=5`。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ update t set d=100 where d=5;		
T2		update t set d=5 where id=0; update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		
T4			insert into t values(1,1,5); update t set c=5 where id=1;
T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 3 假设只在 id=5 这一行加行锁 -- 数据一致性问题

update 的加锁语义和 select ...for update 是一致的，所以这时候加上这条 update 语句也很合理。session A 声明说“要给 d=5 的语句加上锁”，就是为了要更新数据，新加的这条 update 语句就是把它认为加上了锁的这一行的 d 值修改成了 100。

现在，我们来分析一下图 3 执行完成后，数据库里会是什么结果。

1. 经过 T1 时刻，id=5 这一行变成 (5,5,100)，当然这个结果最终是在 T6 时刻正式提交的；
2. 经过 T2 时刻，id=0 这一行变成 (0,5,5)；
3. 经过 T4 时刻，表里面多了一行 (1,5,5)；
4. 其他行跟这个执行序列无关，保持不变。

这样看，这些数据也没啥问题，但是我们再来看看这时候 binlog 里面的内容。

1. T2 时刻，session B 事务提交，写入了两条语句；
2. T4 时刻，session C 事务提交，写入了两条语句；
3. T6 时刻，session A 事务提交，写入了 update t set d=100 where d=5 这条语句。

我统一放到一起的话，就是这样的：

```

1 update t set d=5 where id=0; /*(0,0,5)*/
2 update t set c=5 where id=0; /*(0,5,5)*/
3
4 insert into t values(1,1,5); /*(1,1,5)*/
5 update t set c=5 where id=1; /*(1,5,5)*/
6
7 update t set d=100 where d=5; /* 所有 d=5 的行, d 改成 100*/

```

好，你应该看出问题了。这个语句序列，不论是拿到备库去执行，还是以后用 binlog 来克隆一个库，这三行的结果，都变成了 (0,5,100)、(1,5,100) 和 (5,5,100)。

也就是说，id=0 和 id=1 这两行，发生了数据不一致。这个问题很严重，是不行的。

到这里，我们再回顾一下，**这个数据不一致到底是怎么引入的？**

我们分析一下可以知道，这是我们假设 “select \* from t where d=5 for update 这条语句只给 d=5 这一行，也就是 id=5 的这一行加锁” 导致的。

所以我们认为，上面的设定不合理，要改。

那怎么改呢？我们把扫描过程中碰到的行，也都加上写锁，再来看看执行效果。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ update t set d=100 where d=5;		
T2		update t set d=5 where id=0; (blocked) update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		
T4			insert into t values(1,1,5); update t set c=5 where id=1;
T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 4 假设扫描到的行都被加上了行锁

由于 session A 把所有的行都加了写锁，所以 session B 在执行第一个 update 语句的时候就被锁住了。需要等到 T6 时刻 session A 提交以后，session B 才能继续执行。

这样对于 id=0 这一行，在数据库里的最终结果还是 (0,5,5)。在 binlog 里面，执行序列是这样的：

```
1 insert into t values(1,1,5); /*(1,1,5)*/
2 update t set c=5 where id=1; /*(1,5,5)*/
3
4 update t set d=100 where d=5; /* 所有 d=5 的行，d 改成 100*/
5
6 update t set d=5 where id=0; /*(0,0,5)*/
7 update t set c=5 where id=0; /*(0,5,5)*/
```

[复制代码](#)

可以看到，按照日志顺序执行，id=0 这一行的最终结果也是 (0,5,5)。所以，id=0 这一行的问题解决了。

但同时你也可以看到，id=1 这一行，在数据库里面的结果是 (1,5,5)，而根据 binlog 的执行结果是 (1,5,100)，也就是说幻读的问题还是没有解决。为什么我们已经这么“凶残”地，把所有的记录都上了锁，还是阻止不了 id=1 这一行的插入和更新呢？

原因很简单。在 T3 时刻，我们给所有行加锁的时候，id=1 这一行还不存在，不存在也就加不上锁。

**也就是说，即使把所有的记录都加上锁，还是阻止不了新插入的记录，这也是为什么“幻读”会被单独拿出来解决的原因。**

到这里，其实我们刚说明完文章的标题：幻读的定义和幻读有什么问题。

接下来，我们再看看 InnoDB 怎么解决幻读的问题。

## 如何解决幻读？

现在你知道了，产生幻读的原因是，行锁只能锁住行，但是新插入记录这个动作，要更新的是记录之间的“间隙”。因此，为了解决幻读问题，InnoDB 只好引入新的锁，也就是间隙锁 (Gap Lock)。

顾名思义，间隙锁，锁的就是两个值之间的空隙。比如文章开头的表 t，初始化插入了 6 个记录，这就产生了 7 个间隙。

0	5	10	15	20	25	
$(-\infty, 0)$	$(0, 5)$	$(5, 10)$	$(10, 15)$	$(15, 20)$	$(20, 25)$	$(25, +\infty)$

图 5 表 t 主键索引上的行锁和间隙锁

这样，当你执行 `select * from t where d=5 for update` 的时候，就不止是给数据库中已有的 6 个记录加上了行锁，还同时加了 7 个间隙锁。这样就确保了无法再插入新的记录。

也就是说这时候，在一行行扫描的过程中，不仅将给行加上了行锁，还给行两边的空隙，也加上了间隙锁。

现在你知道了，数据行是可以加上锁的实体，数据行之间的间隙，也是可以加上锁的实体。但是间隙锁跟我们之前碰到过的锁都不太一样。

比如行锁，分成读锁和写锁。下图就是这两种类型行锁的冲突关系。

	读锁	写锁
读锁	兼容	冲突
写锁	冲突	冲突

图 6 两种行锁间的冲突关系

也就是说，跟行锁有冲突关系的是“另外一个行锁”。

但是间隙锁不一样，跟间隙锁存在冲突关系的，是“往这个间隙中插入一个记录”这个操作。间隙锁之间都不存在冲突关系。

这句话不太好理解，我给你举个例子：



session A	session B
begin; select * from t where c=7 lock in share mode;	
	begin; select * from t where c=7 for update;

图 7 间隙锁之间不互锁

这里 session B 并不会被堵住。因为表 t 里并没有 c=7 这个记录，因此 session A 加的是间隙锁 (5,10)。而 session B 也是在这个间隙加的间隙锁。它们有共同的目标，即：保护这个间隙，不允许插入值。但，它们之间是不冲突的。

间隙锁和行锁合称 next-key lock，每个 next-key lock 是前开后闭区间。也就是说，我们的表 t 初始化以后，如果用 select \* from t for update 要把整个表所有记录锁起来，就形成了 7 个 next-key lock，分别是  $(-\infty, 0]$ 、 $(0, 5]$ 、 $(5, 10]$ 、 $(10, 15]$ 、 $(15, 20]$ 、 $(20, 25]$ 、 $(25, +\infty]$ 。

备注：这篇文章中，如果没有特别说明，我们把间隙锁记为开区间，把 next-key lock 记为前开后闭区间。

你可能会问说，这个 suprenum 从哪儿来的呢？

这是因为  $+\infty$  是开区间。实现上，InnoDB 给每个索引加了一个不存在的最大值 suprenum，这样才符合我们前面说的“都是前开后闭区间”。

**间隙锁和 next-key lock 的引入，帮我们解决了幻读的问题，但同时也带来了一些“困扰”。**

在前面的文章中，就有同学提到了这个问题。我把他的问题转述一下，对应到我们这个例子的表来说，业务逻辑这样的：任意锁住一行，如果这一行不存在的话就插入，如果存在这一行就更新它的数据，代码如下：

```
1 begin;
2 select * from t where id=N for update;
3
4 /* 如果行不存在 */
5 insert into t values(N,N,N);
6 /* 如果行存在 */
7 update t set d=N set id=N;
8
9 commit;
```

[复制代码](#)

可能你会说，这个不是 `insert ... on duplicate key update` 就能解决吗？但其实在有多个唯一键的时候，这个方法是不能满足这位提问同学的需求的。至于为什么，我会在后面的文章中再展开说明。

现在，我们就只讨论这个逻辑。

这个同学碰到的现象是，这个逻辑一旦有并发，就会碰到死锁。你一定也觉得奇怪，这个逻辑每次操作前用 `for update` 锁起来，已经是最严格的模式了，怎么还会有死锁呢？

这里，我用两个 session 来模拟并发，并假设  $N=9$ 。

session A	session B
<code>begin;</code> <code>select * from t where id=9 for update;</code>	
	<code>begin;</code> <code>select * from t where id=9 for update;</code>
	<code>insert into t values(9,9,9);</code> <b>(blocked)</b>
<code>insert into t values(9,9,9);</code> <b>(ERROR 1213 (40001): Deadlock found)</b>	

图 8 间隙锁导致的死锁

你看到了，其实都不需要用到后面的 `update` 语句，就已经形成死锁了。我们按语句执行顺序来分析一下：

1. session A 执行 `select ... for update` 语句，由于 `id=9` 这一行并不存在，因此会加上间隙锁 (5,10)；
2. session B 执行 `select ... for update` 语句，同样会加上间隙锁 (5,10)，间隙锁之间不会冲突，因此这个语句可以执行成功；
3. session B 试图插入一行 (9,9,9)，被 session A 的间隙锁挡住了，只好进入等待；
4. session A 试图插入一行 (9,9,9)，被 session B 的间隙锁挡住了。

至此，两个 session 进入互相等待状态，形成死锁。当然，InnoDB 的死锁检测马上就发现了这对死锁关系，让 session A 的 `insert` 语句报错返回了。

你现在知道了，**间隙锁的引入，可能会导致同样的语句锁住更大的范围，这其实是影响了并发度的**。其实，这还只是一个简单的例子，在下一篇文章中我们还会碰到更多、更复杂的例子。

你可能会说，为了解决幻读的问题，我们引入了这么一大串内容，有没有更简单一点的处理方法呢。

我在文章一开始就说过，如果没有特别说明，今天和你分析的问题都是在可重复读隔离级别下的，间隙锁是在可重复读隔离级别下才会生效的。所以，你如果把隔离级别设置为读提交的话，就没有间隙锁了。但同时，你要解决可能出现的数据和日志不一致问题，需要把 binlog 格式设置为 row。这，也是现在不少公司使用的配置组合。

前面文章的评论区有同学留言说，他们公司就使用的是读提交隔离级别加 `binlog_format=row` 的组合。他曾问他们公司的 DBA 说，你为什么要这么配置。DBA 直接答复说，因为大家都这么用呀。

所以，这个同学在评论区就问说，这个配置到底合不合理。

关于这个问题本身的答案是，如果读提交隔离级别够用，也就是说，业务不需要可重复读的保证，这样考虑到读提交下操作数据的锁范围更小（没有间隙锁），这个选择是合理的。

但其实我想说的是，配置是否合理，跟业务场景有关，需要具体问题具体分析。

但是，如果 DBA 认为之所以这么用的原因是“大家都这么用”，那就有问题了，或者说，迟早会出问题。

比如说，大家都用读提交，可是逻辑备份的时候，`mysqldump` 为什么要把备份线程设置成可重复读呢？（这个我在前面的文章中已经解释过了，你可以再回顾下第 6 篇文章 [《全局锁和表锁：给表加个字段怎么有这么多阻碍？》](#) 的内容）

然后，在备份期间，备份线程用的是可重复读，而业务线程用的是读提交。同时存在两种事务隔离级别，会不会有问题？

进一步地，这两个不同的隔离级别现象有什么不一样的，关于我们的业务，“用读提交就够了”这个结论是怎么得到的？

如果业务开发和运维团队这些问题都没有弄清楚，那么“没问题”这个结论，本身就是有问题的。

## 小结

今天我们从上一篇文章的课后问题说起，提到了全表扫描的加锁方式。我们发现即使给所有的行都加上行锁，仍然无法解决幻读问题，因此引入了间隙锁的概念。

我碰到过很多对数据库有一定了解的业务开发人员，他们在设计数据表结构和业务 SQL 语句的时候，对行锁有很准确的认识，但却很少考虑到间隙锁。最后的结果，就是生产库上会经常出现由于间隙锁导致的死锁现象。

行锁确实比较直观，判断规则也相对简单，间隙锁的引入会影响系统的并发度，也增加了锁分析的复杂度，但也有章可循。下一篇文章，我就会为你讲解 InnoDB 的加锁规则，帮你理顺这其中的“章法”。

作为对下一篇文章的预习，我给你留下一个思考题。

session A	session B	session C
begin; select * from t where c>=15 and c<=20 order by c desc for update;		
	insert into t values(11,11,11);	
		insert into t values(6,6,6);

图 9 事务进入锁等待状态

如果你之前没有了解过本篇文章的相关内容，一定觉得这三个语句简直是风马牛不相及。但实际上，这里 session B 和 session C 的 insert 语句都会进入锁等待状态。

你可以试着分析一下，出现这种情况的原因是什么？

这里需要说明的是，这其实是我在下一篇文章介绍加锁规则后才能回答的问题，是留给你作为预习的，其中 session C 被锁住这个分析是有点难度的。如果你没有分析出来，也不要气馁，我会在下一篇文章和你详细说明。

你也可以说说，你的线上 MySQL 配置的是什么隔离级别，为什么会这么配置？你有没有碰到什么场景，是必须使用可重复读隔离级别的呢？

你可以把你的碰到的场景和分析写在留言区里，我会在下一篇文章选取有趣的评论跟大家一起分享和分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

## 上期问题时间

我们在本文的开头回答了上期问题。有同学的回答中还说明了读提交隔离级别下，在语句执行完成后，是只有行锁的。而且语句执行完成后，InnoDB 就会把不满足条件的行行锁去掉。

当然了，c=5 这一行的行锁，还是会等到 commit 的时候才释放的。

评论区留言点赞板：

@薛畅、@张永志同学给出了正确答案。而且提到了在读提交隔离级别下，是只有行锁的。

@帆帆帆帆帆帆帆帆、@欧阳成 对上期的例子做了验证，需要说明一下，需要在启动配置里面增加 `performance_schema=on`，才能用上这个功能，`performance_schema` 库里的表才有数据。



# MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇  
前阿里资深技术专家



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

©版权归极客邦科技所有，未经许可不得转载

上一篇 19 | 为什么我只查一行的语句，也执行这么慢？

写留言

## 精选留言



慧鑫coming

这篇需要多读几遍，again

2018-12-28

👍 2



小新

这篇文章真的需要多啃几遍，

2018-12-28

👍 0



往事随风, 顺其自然

👍 0

总结：for update 是锁住所有行还有间隙锁，但是间隙🔒之间互不冲突，但是互不冲突，为什么插入9这一行会被间隙锁等待，原来没有这一行，这和查询9这一行不是一样？

2018-12-28



Justin

👍 0

下一章老师会不会讲走普通索引，锁普通索引的时候，主键索引，以及其他索引的加锁顺序或者规则呢？很是好奇

2018-12-28