MySQL 是怎么做并发控制的?

马国庆（翊云） 阿里云开发者 2024年10月10日 08:31 浙江

阿里妹导读

本文以 MySQL 8.0.35 的代码为例，尝试对 MySQL 中的并发访问控制进行一个整体的介绍。

前言

最开始学习数据库的时候都会被问到一个问题：“数据库系统相比与文件系统最大的优势是什么？”。具体的优势有很多，其中一个很重要的部分是：数据库系统能够进行更好的并发访问控制。那么，数据库系统到底是怎么进行并发访问控制的？本文以 MySQL 8.0.35 的代码为例，尝试对 MySQL 中的并发访问控制进行一个整体的介绍。

总体介绍

按照近些年流行的概念来讲，MySQL 是一个典型的存储计算分离的架构，MySQL Server 作为计算层，Storage Engine 作为存储层。所以并发访问的控制也需要在计算层和存储层分别进行处理。这里多说一句，MySQL 在设计之初就支持多存储引擎，这也是 MySQL 快速流行的一个很重要的原因，只是随着 MySQL 的发展，到 MySQL 8.0 时代，基本变成了 InnoDB 一家独大的情况。所以本文后续的分析，主要都是围绕 InnoDB 引擎展开。

从数据访问的角度，用户视角下，MySQL 的数据分为：表、行、列。MySQL 内部视角下则包括了：表、表空间、索引、B+tree、页、行、列等。在 MySQL 8.0 中，默认情况下一个表独占一个

表空间，所以为了描述简单，本文后续内容对表和表空间不做区分。

回到主题，MySQL 中的并发访问控制也是基于 MySQL 内部的数据结构来进行设计的，具体包括：

1. 表级别的并发访问控制，包括 Server 层和 Engine 层上的表；
2. 页级别的并发访问控制，包括 Index 和 Page 上的并发访问；
3. 行级别的并发访问控制；

本文后续内容将分为以上三个部分分别展开。

表级别的并发访问控制

我的 DDL 会锁表吗?

在使用数据库的过程中，一个绕不开的操作就是 DDL，特别是在线上运行的库上直接进行 DDL 操作。MySQL 的用户经常会疑惑的一个问题就是：“我这个 DDL 会不会锁表啊？别把业务搞挂了。”之所以会有这样的疑问，是因为在早期的 MySQL 版本中（5.6 之前），DDL 期间是无法进行 DML 操作的，这就导致如果是对一个大表进行 DDL 操作的话，业务会长期无法进行数据写入。为了减少 DDL 期间对业务的应用，衍生出了很多三方的 DDL 工具，其中使用最多的一个是 pt-online-schema-change。

实际上，从 MySQL 5.6 版本开始，MySQL 已经支持 Online DDL 操作；到 5.7 版本，Online DDL 的支持范围进一步扩大，到了 8.0 版本，MySQL 官方进一步支持了 Instant DDL 功能，在 MySQL 上执行 DDL 基本上不会造成业务影响。

关于 Online DDL 的详细介绍，可以直接阅读官方文档[1]，想看精简版的同学，可以参考笔者之前整理的一篇文章[2]。

MDL 锁

DDL 是否会锁表其实就是表级别并发访问控制中最重要的一个问题。MySQL 中实现 DDL、DML、DQL 并发访问最重要的结构就是 MDL 锁。先看一个简单的例子：

```
CREATE TABLE `t1` (
  `id` int NOT NULL,
  `c1` int DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;

INSERT INTO t1 VALUES (1, 10);
INSERT INTO t1 VALUES (2, 20);
INSERT INTO t1 VALUES (3, 30);
```

```
10
```

在上述例子中：

1. session 1 上模拟了一个慢查询；
2. session 2 上执行了一个添加的 DDL，因为查询没有结束，所以 DDL 被阻塞；
3. session 3 上继续进行了查询，查询也会被阻塞，用户觉得“锁表”了；

为什么会出现上述的情况？这里结合 performance_schema 下的 metadata_locks 表可以很清楚的看到等待关系：

可以看到：

1. session 1（THREAD_ID = 57）持有了表上的 SHARED_READ 锁；
2. session 2（THREAD_ID = 58）持有了表上的 SHARED_UPGRADABLE 锁，需要申请表上的 EXCLUSIVE 锁，被阻塞；
3. session 3（THREAD_ID = 59）需要申请表上的 SHARED_READ 锁，被阻塞；

从代码路径上，MDL 的加锁逻辑在打开表的过程中，具体的入口函数为：
open_and_process_table，具体的函数堆栈如下：

```
|--> open_and_process_table
|    |--> open_table
|    |    |--> mdl_request.is_write_lock_request
```

```
|    |    |--> thd->mdl_context.acquire_lock // 请求 global MDL 锁
|    |    |
|    |    |--> open_table_get_mdl_lock
|    |    |    |--> thd->mdl_context.acquire_lock // 请求 table MDL 锁

```

DDL 过程中升级 MDL 锁逻辑的入口函数为mysql_alter_table，具体的函数堆栈如下：

```
|--> mysql_alter_table
|    |--> mysql_inplace_alter_table
|    |    |--> wait_while_table_is_used
|    |    |    |--> thd->mdl_context.upgrade_shared_lock // 升级 MDL 锁
|    |    |    |    |--> acquire_lock // 请求 table MDL EXCLUSIVE 锁

```

通过上面一个简单的例子，我们知道了 MDL 锁的基本概念，也知道了所谓的 DDL 导致“锁表”的原因，严格的说，MDL 锁并不是表锁，而是元数据锁，关于 MDL 更深入的介绍，可以参考这篇文章[3]，本文不再过多展开。MySQL 在 5.6 版本中引入了 MDL 锁，那么是不是有了 MDL 锁之后，其他的表锁就不需要了？

Server 层的表锁

回答上面的问题前，先看一下 MySQL Server 层处理表锁的基本过程。MySQL 中任意表上的操作都需要加表锁，具体的入口函数为lock_tables，具体的函数堆栈如下：

```
|--> lock_tables
|    |--> mysql_lock_tables
|    |    |--> lock_tables_check  // 判断是否需要加锁
|    |    |--> get_lock_data  // 计算有多少张表需要加锁，初始化 MYSQL_LOCK 结构
|    |    |    |--> file->lock_count
|    |    |
|    |    |--> lock_external
|    |    |    |--> ha_external_lock  // 调用 engine handler 接口
|    |    |
|    |    |--> thr_multi_lock
|    |    |    |--> sort_locks
|    |    |    |--> // 遍历加锁
|    |    |    |--> thr_lock  // 加锁 or 等待
|    |    |    |    |--> wait_for_lock // 锁等待，Waiting for table level lock

```

通过上面的堆栈可以看到，整个加锁的过程包括了以下步骤：

1. 加锁前需要先判断对应的表是否需要加锁；
2. 加锁时，需要先调用 Engine 层的 handler 接口加锁；
3. 如果需要，再在 Server 层进行加锁；

对于 InnoDB 引擎，lock_count接口直接返回 0，表示 InnoDB 引擎的表不需要 Server 层后续再加表锁，直接在 external_lock接口中完成所有的处理，这部分后面展开。对于其他引擎，以 CSV 引擎为例，lock_count接口返回 1，所以需要进入到后续的 thr_lock加锁逻辑中。关于thr_lock加锁的类型，以及不同类型锁的冲突关系，此处不再做展开。

狭义上来说，thr_lock接口加的锁就是 Server 层的表锁，具体的加锁逻辑、锁类型的互斥关系、锁等待的逻辑此处不再展开，有兴趣的同学可以自己结合代码进行查看。

InnoDB 中的表锁

前面提到，Server 层的lock_tables接口会调用 Engine 层的 Handler 接口，具体的会调用external_lock接口，那么 InnoDB 在该接口内会去加表锁吗？先看一下函数调用堆栈：

```
|--> ha_innobase::external_lock
|    |--> // lock_type == F_WRLCK
|    |--> m_prebuilt->select_lock_type = LOCK_X
|    |
|    |--> // lock_type == F_RDLCK && trx->isolation_level == TRX_ISO_SERIALIZ
|    |--> m_prebuilt->select_lock_type = LOCK_S
|    |
|    |--> // others
|    |--> m_prebuilt->select_lock_type == LOCK_NONE

|--> row_search_mvcc
|    |--> lock_table(..., prebuilt->select_lock_type == LOCK_S ? LOCK_IS : LO
```

通过上面的堆栈可以看到，进入到 InnoDB 层的加锁逻辑时:

1. 只会先设置后续查询需要的锁类型；
2. 普通的查询操作设置为 LOCK_NONE，后续查询过程无需上锁；
3. 更新操作设置为 LOCK_X，后续查询过程中需要加表上的 IX 锁；

关于 InnoDB 层表锁的具体类型，以及不同类型锁的冲突关系，此处不再做展开。Engine 层的表锁情况，可以在 performance_schema 下的 data_locks 表中进行查看：

LOCK TABLES 操作

前面已经介绍了 MySQL 中的 MDL 锁以及 Server 层和 InnoDB 层的表锁，那么对应到 LOCK TABLES 操作上，到底加的是什么锁？先看一下 LOCK TABLES 操作的执行路径：

```
|--> mysql_execute_command
|    |--> // switch (lex->sql_command)
|    |--> // SQLCOM_LOCK_TABLES
|    |--> trans_commit_implicit // 隐式提交之前的事务
|    |--> thd->locked_tables_list.unlock_locked_tables // 释放之前的表锁
|    |--> thd->mdl_context.release_transactional_locks // 释放之前的 MDL 锁
|    |
|    |--> lock_tables_precheck
|    |--> lock_tables_open_and_lock_tables
|    |    |--> open_tables
|    |    |    |--> lock_table_names // 根据表名加锁（此时还没有打开表）
|    |    |    |    |--> mdl_requests.push_front
|    |    |    |    |--> thd->mdl_context.acquire_locks
|    |    |    |
|    |    |    |--> open_and_process_table
|    |    |
|    |    |--> lock_tables

```

从上面的堆栈可以看到，对于显式的 LOCK TABLES 操作：

1. 会首先隐式提交之前的事务，并且释放掉之前所有的表锁和 MDL 锁；
2. 在打开表之前，直接根据表名进行加锁（如果有其他事务未提交，可能会卡在这里）；
3. 然后进入到正常的打开表和加锁的逻辑；

用一个表格总结一下不同的 LOCK TABLES 操作的加锁情况（InnoDB 表）：

典型线上问题

关于 MySQL 中由于表锁导致的问题，举两个线上常见的案例：

1. DDL 操作导致的 MDL 锁等待。也就是前面在介绍 MDL 锁时举到的例子。其实这类是比较好发现的，直接执行 show processlist 就能看到大量的 MDL 锁等待，这里主要是说明一下如何处理此类问题。处理的方法主要有两种：

   a. 借助performance_schema下的metadata_locks表，找到具体的 MDL 等待关系，然后进行处理(例如：kill 掉慢查询)；

   b. 但是线上多数情况下并没有开启 performance_schema（担心有性能影响），所以也无法从 metadata_locks 表中查询到 MDL 等待关系。此时可以采用另一个方法：直接根据 Time 列进行排序（逆序），然后依次 kill 连接，直到锁等待关系解除。当然，也可以直接 kill 掉所有连接。

2. Server 层表锁导致的性能问题。典型的场景就是开启了 general_log，并且设置输出格式为 TABLE。由于 genelog_log 表是 CSV 引擎，所以需要通过 Server 层的表锁来控制并发插入，当写入量很大时，CSV 表的写入会出现性能瓶颈。从现象上看，就是大量的连接等待表锁 “Waiting for table level lock”。CSV 表的写性能问题暂时没有好的优化方式，所以遇到之后最好的处理手段就是直接关闭 general_log。

表级别的加锁过程总结

以上就是表级别的加锁过程，做一个总结：

1. 最先加的是 MDL 锁，在打开表时（open_and_process_table接口）就需要根据操作的类型确定 MDL 的锁类型（实际上，大部分请求在词法解析阶段就已经完成了 MDL 请求的初始化）；
2. 在实际的 SQL 操作时，会根据操作的类型，在不同的位置调用 lock_tables接口加表锁，表锁又分为 Server 层的表锁和 Engine 层的表锁：
   a. 对于 InnoDB 引擎，直接调用 Engine 层的 external_lock接口去加 Engine 层的表锁（通过前面的代码堆栈知道，其实只是确定后续需要加锁的类型，加锁动作是后置的），不需要再在 Server 层加表锁；
   b. 对于 CSV 引擎，Engine 层并没有实现 external_lock接口，所以需要在 Server 层加表锁；

页级别的并发访问控制

B+tree 的基本结构

InnoDB 引擎通过 B+tree 来保存数据，关于 B+tree 介绍的文章网上有很多，大家可以自行查询学习。这里只是简单介绍一下 B+tree 的基本结构，方便后续的描述。

上图所示是一个典型的三层结构的 B+tree，其中：

1. 最上层的为根节点（ROOT），每个 B+tree 都只会有一个根节点；
2. 最下层的为叶子节点（LEAF），叶子节点也是实际保存数据的节点；
3. 中间层为非叶子节点（根节点也其实也是非叶子节点），保存索引数据，根据 B+tree 本身的大小，可能有 0 到多个中间层；

从上图中可以看到，层与层之间有一个单向的指针（上层到下层），层之间不同节点间有一个双向的指针。B+tree 中的每一个节点都是一个数据页（Page），页也是 InnoDB 中数据读写的最小单元，InnoDB 中默认的页大小为 16KB。

对于 InnoDB 表，经常听到一个概念叫做“索引组织树”，笔者理解的意思就是每张 InnoDB 表的每一个索引都是一棵 B+tree，数据就保存在 B+tree 上。关于 InnoDB 中更多索引的概念，包括：主键索引、二级索引、聚簇索引、覆盖索引等等，不是本文讨论的重点，所以此处不再展开，感兴趣的读者可以自行查询学习。

B+tree 的加锁过程

页级别的并发访问控制主要通过 index 和 page 上的锁来实现，其实也就是 B+tree 的加锁过程。在介绍加锁过程前，先结合数据看一下 B+tree 的访问路径。

在之前的 B+tree 结构上补充了主键信息（ID），假设现在需要访问的数据是 ID = 400 的行，那么 B+tree 上的访问路径如上图所示。可以看到，首先访问根节点，然后根据主键找到下一层的非叶子节点，然后继续向下找到对应的叶子节点，读取数据。

事实上，B+tree 的加锁过程其实也是按照上述访问路径进行的。还是以上述的查询为例，B+tree 上加锁的过程如下图所示：

具体的步骤如下：

1. 加 index 上的 S 锁；
2. 加根节点上的 S 锁；

3. 加非叶子节点上的 S 锁；
4. 加叶子节点上的 S 锁；
5. 释放 index 和所有非叶子节点上的 S 锁；

类似的，如果是页上的乐观更新（或者是页内的插入），那么 B+tree 上加锁的过程如下图所示：

具体的步骤如下：

1. 加 index 上的 S 锁；
2. 加根节点上的 S 锁；
3. 加非叶子节点上的 S 锁；
4. 加叶子节点上的 X 锁；
5. 释放 index 和所有非叶子节点上的 S 锁；

可以看到，如果是页内的修改，其实加锁的逻辑和读过程的加锁类似很像，只是最后在叶子节点上加锁的类型不一样。

SMO 问题

上面介绍了查询过程和发生页内修改时 B+tree 上的加锁过程，如果更新的数据无法在页内完成，或者说修改动作会造成 B+tree 结构的变化（SMO, Structure Modify Operation），又应该如何进行加锁?

InnDB 在执行数据更新操作时，会首先尝试使用乐观更新（MODIFY LEAF），如果乐观更新失败，那么会 进入到悲观更新（MODIFY TREE）的逻辑，悲观更新的加锁过程如下图所示：

具体的步骤如下：

1. 加 index 上的 SX 锁；
2. 根节点不加锁
3. 非叶子节点上不加锁，但是会搜索所有经过的节点；
4. 判断可能修改的非叶子节点加 X 锁，根节点加 SX 锁；
5. 叶子节点，包括前后叶子节点加 X 锁；

可以看到，和前面不同的是，进入到悲观更新的逻辑时，会直接对 index 加 SX 锁（在 5.7 之前的版本中是直接加 X 锁，5.7 版本引入了 SX 锁，SX 锁和 S 锁不互斥，所以此时还可以读），所以在后续 B+tree 遍历的过程中，只是先收集索引经过的节点，并没有直接上锁。只有到了要修改的叶子节点时，才会去判断哪些非叶子节点也可能会修改，从而加上 X 锁。

所以在整个 SMO 期间，除了可能会被修改的叶子节点和非叶子节点加的是 X 锁之外，其他的节点都没有加锁（index 和根节点是 SX 锁），非修改节点上的读操作可以正常进行。但是一棵 B+tree 上同时只能有一个 SMO 操作。

一个写入过程中的 B+tree 加锁过程

整个 B+tree 加锁的过程比较复杂，这里以一个主键上的插入过程对主要的代码堆栈进行说明：

```
|--> row_ins_clust_index_entry
|    |--> row_ins_clust_index_entry_low(..., BTR_MODIFY_LEAF, ...) // 乐观
|    |    |--> pcur.open(index, ...)
```

```
|    |    |    |--> btr_cur_search_to_nth_level // 遍历 b+tree
|    |    |    |    |--> // switch (latch_mode)
|    |    |    |    |--> // default
|    |    |    |    |--> mtr_s_lock(dict_index_get_lock(index), ...) // index
|    |    |    |    |--> btr_cur_latch_for_root_leaf
|    |    |    |    |
|    |    |    |    |--> // search_loop
|    |    |    |    |--> // retry_page_get
|    |    |    |    |--> buf_page_get_gen(..., rw_latch, ...)
|    |    |    |    |    |--> mtr_add_page // 按类型对 page 加锁
|    |    |
|    |--> row_ins_clust_index_entry_low(..., BTR_MODIFY_TREE, ...) // 悲观
|    |    |--> pcur.open(index, ...)
|    |    |    |--> btr_cur_search_to_nth_level // 遍历 b+tree
|    |    |    |    |--> // switch (latch_mode)
|    |    |    |    |--> // BTR_MODIFY_TREE
|    |    |    |    |--> mtr_sx_lock(dict_index_get_lock(index), ...) // index
|    |    |    |    |--> btr_cur_latch_for_root_leaf
|    |    |    |    |
|    |    |    |    |--> // search_loop
|    |    |    |    |--> // retry_page_get
|    |    |    |    |--> buf_page_get_gen(..., rw_latch, ...)
|    |    |    |    |    |--> mtr_add_page // 按类型对 page 加锁
```

以上只是 B+tree 加锁过程的一个入口介绍，详细的加锁逻辑可以通过上述入口自行进行展开，再此不做进一步的展开。

B+tree 加锁过程总结

以上就是 B+tree 的加锁过程，做一个总结：

1. 页级别的并发访问控制发生在 B+tree 的遍历过程，也就是 B+tree 的加锁过程；
2. 加锁的对象包括了 index 和 page；
3. 加锁的类型包括了 S，SX 和 X，其中 S 锁和 SX 锁不互斥；
4. 查询过程只加 S 锁；
5. 修改过程，根据修改的类型加锁过程有所区别。如果是页内的数据修改，走乐观更新的逻辑，只有被修改的叶子节点加 X 锁；如果是悲观更新的逻辑，index 和根节点要加 SX 锁，索引可能被修改的节点都要加 X 锁；

行级别的并发访问控制

一个有趣的死锁问题

在介绍行级别的并发访问控制前，先一起看一个有意思的问题：

```
CREATE TABLE `t1` (
  `id` int NOT NULL,
  `c1` int DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;

INSERT INTO t1 VALUES (1, 10);
INSERT INTO t1 VALUES (2, 20);
INSERT INTO t1 VALUES (3, 30);

```

	session 1	session 2	session 3
T1	BEGIN; INSERT INTO t1 VALUES (4, 40);		
T2		INSERT INTO t1 VALUES (4, 40);	
		-- wait	
T3			INSERT INTO t1 VALUES (4, 40);
			-- wait
T4	ROLLBACK;		
		?	?

在上面的例子中，事务隔离级别默认是 RC（READ-COMMITTED，读已提交）。session 1 开启了一个事务，然后插入了一行数据，并且没有提交。session 2 和 session 3 随后插入了相同的数据，但是都会被阻塞。最后 session 1 进行了回滚操作，那么 session 2 和 session 3 分别会发生什么?

了解 MySQL 行锁原理的同学可能会给出下面的答案：session 2 插入成功，session 3 报错，错误类型是 'Duplicate key'。那么真的是这样吗?

笔者这里直接贴出在 MySQL 8.0.35 上的执行结果。

[session 2] 执行结果：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 19
Server version: 8.0.35-rds-dev-debug Source distribution

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> INSERT INTO t1 VALUES (4, 40);
Query OK, 1 row affected (5.52 sec)
```

[session 3] 执行结果：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 20
Server version: 8.0.35-rds-dev-debug Source distribution

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> INSERT INTO t1 VALUES (4, 40);
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

可以看到，session 2 确实插入成功了，session 3 也报错了，但是这个报错怎么看上去不太符合我们一般的认知，为什么是死锁（Deadlock found），死锁从何而来?

为了搞清楚这个问题，笔者关闭 MySQL 上的死锁检查逻辑（innodb_deadlock_detect设置为OFF），然后再次尝试了上述的操作。结果发现，session 2 和 session 3 确实卡住了，结果前面提到的 performance_schema 下的 data_locks 表进行查看：

```
mysql> select * from data_locks order by THREAD_ID;
+--------+-----------------------------------+-----------------------+-----------+----------+---------------+-------------+----------------+---------------+-----------+-----------------------+-----------+---------------+-------------+-----------------------+
| ENGINE | ENGINE_LOCK_ID                    | ENGINE_TRANSACTION_ID | THREAD_ID | EVENT_ID | OBJECT_SCHEMA | OBJECT_NAME | PARTITION_NAME | SUBPARTITION_NAME | INDEX_NAME | OBJECT_INSTANCE_BEGIN | LOCK_TYPE | LOCK_MODE     | LOCK_STATUS | LOCK_DATA             |
+--------+-----------------------------------+-----------------------+-----------+----------+---------------+-------------+----------------+---------------+-----------+-----------------------+-----------+---------------+-------------+-----------------------+
| INNODB | 140322692159448:4:4:1:140322621423040 |                  1943 |        66 |       36 | my_demo       | t1          | NULL           | NULL          | PRIMARY   |       140322621423040 | RECORD    | S             | GRANTED     | supremum pseudo-record |
| INNODB | 140322692158440:4:4:1:140322621416768 |                  1942 |        66 |       36 | my_demo       | t1          | NULL           | NULL          | PRIMARY   |       140322621416768 | RECORD    | S             | GRANTED     | supremum pseudo-record |
| INNODB | 140322692158440:1066:140322621419584 |                  1942 |        69 |       15 | my_demo       | t1          | NULL           | NULL          | NULL      |       140322621419584 | TABLE     | IX            | GRANTED     | NULL                  |
| INNODB | 140322692158440:4:4:1:140322621417120 |                  1942 |        69 |       15 | my_demo       | t1          | NULL           | NULL          | PRIMARY   |       140322621417120 | RECORD    | X,INSERT_INTENTION | WAITING     | supremum pseudo-record |
| INNODB | 140322692159448:1066:140322621425856 |                  1943 |        70 |       12 | my_demo       | t1          | NULL           | NULL          | NULL      |       140322621425856 | TABLE     | IX            | GRANTED     | NULL                  |
| INNODB | 140322692159448:4:4:1:140322621423392 |                  1943 |        70 |       12 | my_demo       | t1          | NULL           | NULL          | PRIMARY   |       140322621423392 | RECORD    | X,INSERT_INTENTION | WAITING     | supremum pseudo-record |
+--------+-----------------------------------+-----------------------+-----------+----------+---------------+-------------+----------------+---------------+-----------+-----------------------+-----------+---------------+-------------+-----------------------+
6 rows in set (0.00 sec)
```

通过 data_locks 表中的锁等待关系发现，session 2（THREAD_ID = 69）和 session 3（THREAD_ID = 70）都在等待意向锁，隐含的语义是都持有了 Gap 锁，所以造成了死锁。

所以，通过上面的例子可以知道，即使是 RC 隔离级别下简单的主键插入，也并不只是对单行记录加锁，并且还可能造成死锁。

行锁的基本概念

通过上面的例子可以知道，InnoDB 中的行锁并不只是单行记录上的锁，实际上 InnoDB 内部对行锁分为了以下几种：

1. 记录锁（Rec Lock），即对单行记录上加的锁，官方代码中的名字是 LOCK_REC_NOT_GAP；从加锁类型上来说，记录锁优又可以分为记录读锁（S 锁）和记录写锁（X）锁；
2. 间隙锁（Gap Lock ），对行记录的间隙加的锁，官方代码中的名字是 LOCK_GAP；（补充一句，网上有很多文章都说 Gap 锁是为了解决 RR （REPEATABLE-READ，可重复读）隔离级别写的幻读问题，其实并不完全是，前面的插入死锁的例子也能说明。关于事务隔离级别的问题，不是本文讨论的重点，这里不再展开。）
3. 下键锁（Next-Key Lock），可以简单的理解就是记录锁和间隙锁的组合（记录前的间隙），官方代码中的名字是 LOCK_ORDINARY；
4. 插入意向锁（Insert Intention Lock），如果插入的位置已经被别的事务加了 Gap 锁，那么当前插入就需要进行等待，这个时候就会生成一个插入意向锁，官方代码中的名字是 LOCK_INSERT_INTENTION；

以上就是 InnoDB 中行锁的基本概念，看起来非常的简单，但是真正理解并不容易，所以后文主要是通过一些典型的案例来进行分析。

一个写入过程中的加锁过程

在进入案例分析前，还是以一个写入过程为例，结合代码进行一个主要逻辑的说明，主要的代码堆栈如下：

```
|--> ha_innobase::write_row
|    |--> row_insert_for_mysql
|    |    |--> row_insert_for_mysql_using_ins_graph
|    |    |    |--> // run_again
|    |    |    |--> row_ins_step
|    |    |    |    |--> row_ins
|    |    |    |    |    |--> row_ins_index_entry_step
|    |    |    |    |    |    |--> row_ins_index_entry
|    |    |    |    |    |    |    |--> row_ins_clust_index_entry // 插入主键
|    |    |    |    |    |    |    |    |--> row_ins_clust_index_entry_low
|    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |--> row_ins_sec_index_entry // 插入二级索引
|    |    |    |    |
|    |    |    |    |--> row_mysql_handle_errors
|    |    |    |    |    |--> lock_wait_suspend_thread // 锁等待，唤醒后进入 run_again
|
|--> row_ins_clust_index_entry_low
|    |--> btr_pcur_t::open  // 遍历 b+tree
|    |
|    |--> row_ins_duplicate_error_in_clust // 第一次插入不会进入（隐式锁）
|    |    |--> row_ins_set_rec_lock
|    |    |    |--> lock_clust_rec_read_check_and_lock
|    |    |    |    |--> lock_rec_convert_impl_to_expl // 隐式锁转显式锁
|    |    |    |    |    |--> lock_rec_convert_impl_to_expl_for_trx
|    |    |    |    |    |    |--> lock_rec_add_to_queue
```

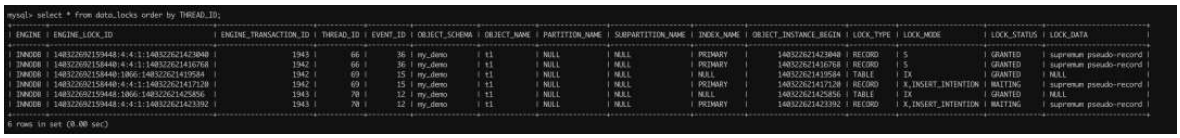
```
|    |    |    |    |    |    |    |    |--> rec_lock.create // RecLock::create
|    |    |    |    |    |    |    |    |    |--> lock_alloc
|    |    |    |    |    |    |    |    |    |--> lock_add
|    |    |    |    |    |    |    |    |    |    |--> // 不等待
|    |    |    |    |    |    |    |    |    |    |--> lock_rec_insert_to_granted
|    |    |    |    |    |    |    |    |    |    |--> locksys::add_to_trx_locks
|    |    |    |    |    |--> lock_rec_lock // 构造锁等待
|    |    |    |    |    |    |--> lock_rec_lock_fast
|    |    |    |    |    |    |    |--> rec_lock.create // RecLock::create
|    |    |    |    |    |    |--> lock_rec_lock_slow
|    |    |    |    |    |    |    |--> lock_rec_has_expl
|    |    |    |    |    |    |    |--> lock_rec_other_has_conflicting // 检查冲突
|    |    |    |    |    |    |    |--> rec_lock.add_to_waitq
|    |    |    |    |    |    |    |    |--> create // RecLock::create
|    |    |    |    |    |    |    |    |    |--> lock_alloc
|    |    |    |    |    |    |    |    |    |--> lock_add
|    |    |    |    |    |    |    |    |    |    |--> // 等待
|    |    |    |    |    |    |    |    |    |    |--> lock_rec_insert_to_waiting
|    |    |    |    |    |    |    |    |    |    |--> locksys::add_to_trx_locks
|    |    |    |    |    |    |    |    |    |    |--> lock_set_lock_and_trx_wait
|    |    |
|    |    |--> btr_cur_optimistic_insert // 乐观插入
|    |    |    |--> btr_cur_ins_lock_and_undo
|    |    |    |    |--> lock_rec_insert_check_and_lock // 插入前的锁冲突检查
|    |    |    |    |    |--> lock_rec_other_has_conflicting
|    |    |    |    |    |--> rec_lock.add_to_waitq
|    |    |    |    |--> trx_undo_report_row_operation
|    |    |--> btr_cur_pessimistic_insert // 悲观插入

```

从上述代码来看，最开始的例子中的插入过程应该是这样的：

1. session 1 进行插入时，因为是第一次插入，所以不需显式的创建锁，直接插入；
2. session 2 进行插入时，在：
   row_ins_duplicate_error_in_clust函数中进行冲突检查时：
   a. 发现记录已经存在，并且对应的事务是一个活跃事务，这个时候会触发隐式锁转显示锁的逻辑，简单来说就是 session 2 为 session 1（准确的说是 trx 1）创建一个 Rec X Lock，因为这个时候还不存在任何等到关系，所以可以直接获取到锁；
   b. 继续为自己创建一个 Rec S Lock，由于和前面的 Rec X Lock 冲突，所以会加入到等待队列，跳过后续的插入操作，最后进入到lock_wait_suspend_thread函数中进行等待；
3. session 3 进行插入时，基本过程和 session 2 是一样的，只是发现 Rec X Lock 已经存在了，不需要再触发隐式锁转显示锁的逻辑，直接为自己创建一个 Rec S Lock，进入等待；

至此，session 2 和 session 3 的等待逻辑已经比较清楚了。但是为什么 session 1 回滚之后，session 2 和 session 3 会形成死锁？按照前面的分析，session 1 回滚释放了 Rec X Lock，session 2 和 session 3 被唤醒，那么应该是哪个线程先进入到插入逻辑，哪个线程插入成功，另一个线程失败。另外，前面提到的，session 2 和 session 3 最终等待在插入意向锁上，理论上这个锁的出现必须要有其他线程持有 Gap 锁，这个等待关系是如何出现的？



这个问题笔者也思考了很长的时间，最后还得通过源码调试的方式找到答案。简单的来说就是：

1. session 1 在回滚的时候，并不是简单的释放 Rec X Lock，然后唤醒 session 2 和 session 3；

2. session 1 的回滚逻辑里面有一个非常重要的步骤lock_rec_inherit_to_gap，该函数会把 session 2 和 session 3 上的 Rec Lock 转换为 Gap Lock；

3. session 2 和 session 3 被唤醒后，不论是哪个线程先进入到插入逻辑，都会在插入前的锁冲突检查中：
   （lock_rec_insert_check_and_lock）发现对方的 Gap 锁，然后生成插入意向锁；

关于写入过程中的加锁过程，上面只是借助 insert 导致死锁的案例进行了一个非常简单的介绍，重点还是说清楚插入过程中行锁的产生以及锁等待产生的基本逻辑，InnoDB 行锁的内容非常丰富，此处不再继续展开，后面有时间可以单独再进行介绍。

典型死锁问题

注：以下场景主要来源于：
https://www.modb.pro/db/1703591734429175808，各个场景中使用的表结构如下：

```
DROP TABLE IF EXISTS `t1`;

CREATE TABLE `t1` (
  `id` int NOT NULL AUTO_INCREMENT,
  `a` int DEFAULT NULL,
  `b` int DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `uk_a` (`a`)
) ENGINE=InnoDB;

INSERT INTO t1 values (1, 10, 0);
INSERT INTO t1 values (2, 20, 0);
INSERT INTO t1 values (3, 30, 0);
INSERT INTO t1 values (4, 40, 0);
```

```
15  INSERT INTO t1 values (5, 50, 0);
16
```

注：需关闭 MySQL 上的死锁检查逻辑（innodb_deadlock_detect设置为 OFF）。

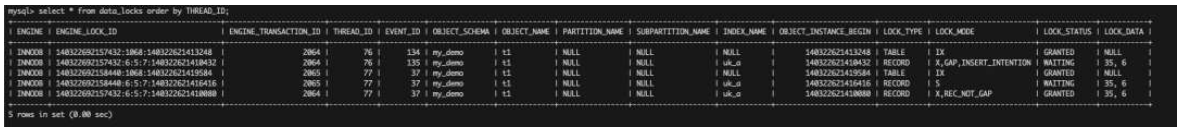
场景 1

	session 1	session 2
T1	BEGIN; INSERT INTO t1(a, b) VALUES (35, 0);	
T2		BEGIN; INSERT INTO t1(a, b) VALUES (35, 0);
		-- wait
T3	INSERT INTO t1(a, b) VALUES (33, 0);	
	-- wait	

T1：session 1 第一次插入，不显式的创建锁；

T2：session 2 插入重复的行，首先为 session 1（trx 1） 创建 UK 上 (30, 35] 的下键锁（X），然后需要为自己创建 UK 上 (30, 35] 的下键锁（S），此时 Gap 锁不冲突，所以 session 最中是等在 35 上的记录锁（S）；

T3：session 1 再次插入，由于插入的区间还是 (30, 35)，和 session 2 持有的区间锁冲突，所以产生一个插入意向锁，最终导致死锁；



场景 2

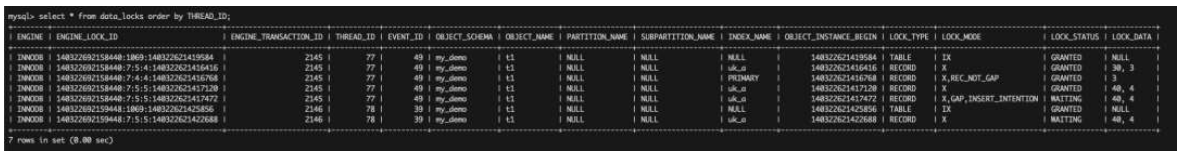
	session 1	session 2	session 3
T1	BEGIN; REPLACE INTO t1(a, b) VALUES (40, 1);		
T2		BEGIN; REPLACE INTO t1(a, b) VALUES (30, 1);	
		-- wait	
T3			BEGIN; REPLACE INTO t1(a, b) VALUES (40, 1);
			-- wait
T4	ROLLBACK;		

T1：session 1 更新时，检测到 UK 冲突，创建 UK 上 (30, 40] 和 (40, 50] 上的下键锁（X）；

T2：session 2 更新时，也检测到 UK 冲突，需要创建 UK 上 (20, 30] 和 (30, 40] 上的下键锁（X），由于 session 1 已经持有了 40 上的记录锁（X），所以 session 2 只能等到 40 上的记录锁（X）；

T3：session 3 更新时，也检测到 UK 冲突，需要创建 UK 上 (30, 40] 和 (40, 50] 上的下键锁（X），由于 session 1 已经持有了 40 上的记录锁（X），所以 session 3 只能等到 40 上的记录锁（X），此时还未开始处理 (40, 50] 上的下键锁（X）；

T4：session 1 回滚，释放锁；session 2 获得 40 上的记录锁（X），但是由于插入的区间是（20,40），且 session 3 已经持有了 (30, 40) 上的 Gap 锁，所以需要产生一个插入意向锁，最终导致死锁；



注意：这里其实有一个很有意思的问题，为什么 UK 上的更新需要加两个下键锁，感兴趣的同学可以参考这篇文章[4]。

场景 3

	session 1	session 2	session 3
T1	BEGIN; SELECT * FROM t1 WHERE a = 40 for UPDATE;		
T2		BEGIN; REPLACE INTO t1(a, b) VALUES (30, 1);	
		-- wait	
T3			BEGIN; REPLACE INTO t1(a, b) VALUES (40, 1);
			-- wait
T4	ROLLBACK;		

T1：session 1 执行查询，由于加了 for UPDATE保护，所以需要加 40 上的记录锁（X）；

后续步骤和场景 2 相同，不再赘述。

场景 4

	session 1	session 2	session 3
T1	BEGIN; INSERT INTO t1(id, a, b) VALUES (6, 60, 0);		
T2		BEGIN; INSERT INTO t1(id, a, b) VALUES (6, 70, 0);	
		-- wait	
T3			BEGIN; INSERT INTO t1(id, a, b) VALUES (6, 70, 0);
			-- wait
T4	ROLLBACK;		

这个场景和前面解释的死锁场景（一个有趣的死锁问题）是一样的，不再重复分析。

死锁问题的排查

上面结合了例子讲了一些行级别的并发访问控制导致的死锁问题，最后简单说一下出现死锁问题的排查思路：

1. MySQL 8.0 默认开启了死锁检测（innodb_deadlock_detect），原则上不建议手动关闭；此外innodb_lock_wait_timeout参数也不建议设置过大；
2. 当出现死锁时，如果开启了 performance_schema，可以通过查询 performance_schema 下的 data_locks 表查看所等待关系，然后手动进行处理；和 MDL 锁等到处理的逻辑类似，如果实在不想分析锁等待的关系，可以把 data_locks 表中所有涉及的连接全部 kill；
3. 如有真的出现了死锁，在 MySQL 的错误日志中会打印出锁等待关系，可以通过锁等待关系进行分析，优化业务侧的写入逻辑；

行级别的加锁过程总结

以上就是行级别的加锁过程，做一个总结：

1. 行锁并不只是行记录上的锁，行锁的类型包括了：记录锁（Rec Lock）、间隙锁（Gap Lock）、下键锁（Next-Key Lock）和插入意向锁（Insert Intention Lock）；
2. 行锁是按需创建的，如果是第一次插入，默认不加锁（隐式锁），只有出现冲突时才会升级为显式锁；

3. 记录锁（Rec Lock）上只有 S 锁和 S 锁兼容；
4. 间隙锁（Gap Lock）上 S 锁和 X 锁可以兼容，X 锁和 X 锁也可以兼容；
5. 下键锁（Next-Key Lock）就是记录锁和间隙锁的组合，处理的时候也是分开的；
6. 插入意向锁（Insert Intention Lock）的产生一定是因为有其他事务持有个待插入间隙的间隙锁；
7. 所有锁的释放都是在事务提交时，所以为了减少死锁的产生，建议事务尽快提交；

总结

本文主要是笔者对 MySQL 中表、页和行上的并发访问控制和加锁逻辑的一个整理，总的来说：

1. 表、页、行其实就是 MySQL 数据处理的基本流程；
2. 表上的并发控制，或者说表锁主要保护的是表结构，在 MySQL 8.0 版本中，表结构的保护都是由 MDL 锁完成；非 InnoDB 表（CSV 表）还会依赖 Server 层的表锁进行并发控制，InnoDB 表不需要 Server 层加表锁；
3. 页上的并发控制，或者说 index 和 page 上的锁主要是为了保护 B+tree 的安全性，乐观写入下，只有叶子节点上需要加 X 锁；悲观写入下（SMO），索引可能修改的节点上都需要加 X 锁。引入 SX 锁增加了读写并发，但是 SMO 操作依然不能并发；
4. 行上的并发控制，或者说行锁主要是为了保护行记录的一致性，其实行上的并发控制还有一个很重要的点是 MVCC，本文没有对这部分内容进行展开，感兴趣的同学可以自行学习；

在写这篇文章之前，关于 MySQL 内部各种锁的介绍文章已经很多了，而且只要是稍微了解数据库，了解 MySQL 的同学其实都会有一个自己对于各种锁的认知。为什么要写这篇文章，一是觉得很多网上的文章都太偏重于概念，一上来就是共享锁与互斥锁，乐观锁与悲观锁，显式锁与隐式锁，要不就是一个表格告诉你各种锁的互斥与兼容关系，而没有结合实际的例子来说明为什么要这么加锁，一看一个不吱声；二是最近刚好碰到了几个线上问题，所以趁此机会把之前分散整理的一些文档统一梳理了一遍，更多的还是自己的理解，如果文档中有描述错误的地方，欢迎批评指正。想了下叫做 MySQL 中的锁分析好像也不合适，所以就改成了 MySQL 是怎么做并发控制的。当然，MySQL 中的并发控制远不止这些，有机会的话后面会继续补充。

参考链接：

[1]https://dev.mysql.com/doc/refman/8.4/en/innodb-online-ddl-operations.html

[2]http://mysql.taobao.org/monthly/2021/03/06/

[3]https://developer.aliyun.com/article/877241

[4]http://mysql.taobao.org/monthly/2022/05/02/

写留言