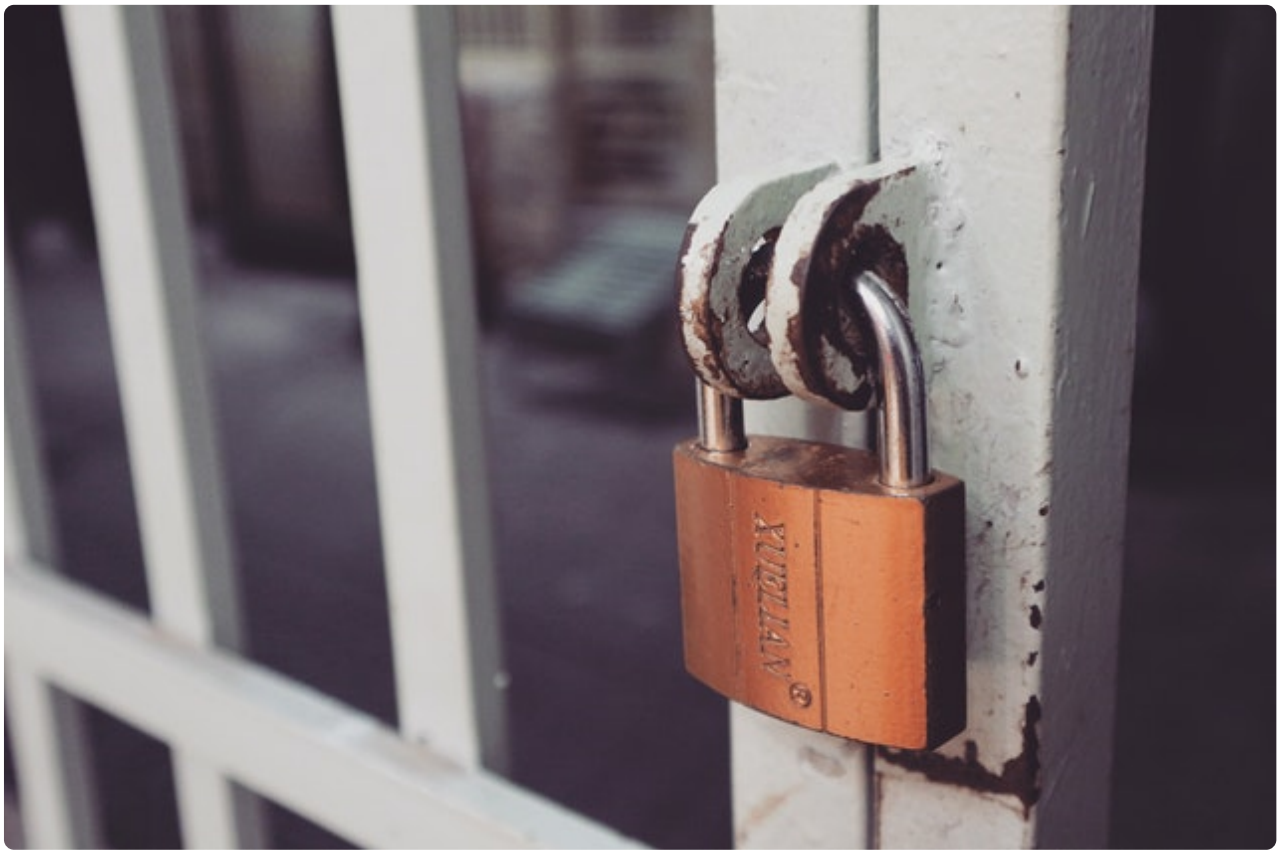
12 死锁是怎么出现的？又是怎么解决的呢？

更新时间：2020-03-27 10:12:59



> 智慧，不是死的默念，而是生的沉思。——斯宾诺莎

死锁这个概念大家应该是很熟悉了，它最早出现于操作系统中，指的是两个进程持有对方想要的资源，但是又都不会释放这些资源，那么，就只能无止境的等待。在 MySQL 中，我们说的死锁是事务相关的，所以，不同的存储引擎死锁的产生条件和解决办法也是不相同的。这一节里，我们就要去看一看在 InnoDB 中，死锁是怎么出现的，又怎样去避免和解决它。

1 你需要知道的死锁理论

这里所涉及的死锁理论不仅仅适用于 InnoDB，同样也适用于操作系统和我们编写的代码。学习这些基本理论是非常重要的，它将会有助于我们去判断（各种应用中）是否发生了死锁，以及指导我们怎样去避免死锁等等。

1.1 什么是死锁

首先，我们先来读一读死锁的定义：

> 当两个以上的运算单元，双方都在等待对方停止运行，以获取系统资源，但是没有一方提前退出时，就称之为死锁。

死锁是计算机系统中的常见问题，之后被引申到类似场景的各种应用中。由于资源占用是互斥的，当某个进程（或事务）提出资源申请后，使得其他进程在无外力协助下，永远分配不到资源而无法继续运行，此时就称系统处于死锁状态或产生了死锁。

1.2 死锁产生的必要条件

任何事件的产生都一定会有其前提条件，死锁自然也不会例外。对于死锁的产生，必须要同时具备以下的四个条件（必要条件）：

- 互斥条件：某个资源在同一时刻只能被一个进程占有
- 不可剥夺条件：一个进程占有的资源，在没有使用完之前，不能被其他的进程抢占
- 请求与保持条件：一个进程因请求资源阻塞时，对自己占据的资源不释放
- 循环等待条件：若干个进程之间形成了一种头尾相接的循环等待资源关系

所谓必要条件，就是要同时满足，那么，我们也可以得到启发：打破死锁的关系，只需要让以上的四个条件不同时满足就可以了。

数据库死锁的影响是非常大的，在生产环境中，几乎是致命的，随时可能会导致系统崩溃。例如，某张表由于各种原因出现了死锁，那么，所有涉及这张表的操作都会被阻塞，不论读写。这就会使很多操作在队列里排队，占用宝贵的数据库连接。最终会导致数据库连接耗尽、各种操作超时等等，致使系统各项指标异常，进而引发系统崩溃。所以，你需要去理解死锁，尽量避免死锁，并在出现死锁后能够快速处理死锁。

2 InnoDB 中出现的死锁

根据之前对死锁的描述（定义及必要条件），我们应该可以自己“制造出”死锁。同时，在工作中，不恰当的使用方法与并发事务引起的死锁也是很常见的。下面，我将会模拟一些死锁的案例，因为你只有知道怎样的操作会引起死锁，才会想办法不做那样的操作。

2.1 满足死锁的必要条件模拟死锁

要让事务的操作过程死锁，就必须同时满足四个条件（你可以想一想，怎样把死锁的四个条件翻译为 InnoDB 死锁的四个条件）。首先，仍然是给出一些示例数据（对，没错，还是那个常见的 worker 表）：

```
mysql> SELECT * FROM worker WHERE id < 3;
+----+------+------+--------+---------+
| id | type | name | salary | version |
+----+------+------+--------+---------+
|  1 | A    | tom  |   1800 |       0 |
|  2 | B    | jack |   2100 |       0 |
+----+------+------+--------+---------+
```

出现死锁，至少需要有两个事务在同时工作，所以，我们需要开启两个 MySQL 客户端，我把它们称之为“会话 A”和“会话 B”。接下来，我要演示一个完整的死锁过程（当然也会附有详细的注释说明），一起看看吧。

```
-- “会话 A” 关闭自动提交
mysql> SET AUTOCOMMIT = off;
Query OK, 0 rows affected (0.00 sec)

-- “会话 A” 开启事务
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

-- “会话 A” 更新 id = 1 的记录（更新什么不重要，重要的是这个更新事务）
mysql> UPDATE worker SET type = 'B' WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

-- “会话 B” 关闭自动提交
mysql> SET AUTOCOMMIT = off;
Query OK, 0 rows affected (0.00 sec)

-- “会话 B” 开启事务
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

-- “会话 B” 更新 id = 2 的记录（更新什么同样是不重要的）
mysql> UPDATE worker SET type = 'A' WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

-- “会话 A” 更新 id = 2 的记录（你会发现事务卡住了）
mysql> UPDATE worker SET type = 'A' WHERE id = 2;

-- “会话 B” 更新 id = 1 的记录（可以看到，出现了死锁，MySQL 报错了，并让我们尝试重启事务）
mysql> UPDATE worker SET type = 'B' WHERE id = 1;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

-- 回到 “会话 A”（注意，我们此时并不做任何操作），看看发生了什么
mysql> UPDATE worker SET type = 'A' WHERE id = 2;
Query OK, 1 row affected (42.99 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

最后，你会发现，“会话 A” 更新 id = 2 的记录执行成功了（注意看执行语句耗时，这是在等待锁），这是因为 “会话 B” 出现了死锁被 MySQL KILL 掉了。所以，MySQL 才会建议我们重新开启事务。

以上就是一个最简单、最典型的 InnoDB 死锁案例，相信你看到这个过程之后，会对死锁有更进一步的理解。那么，得出 InnoDB 死锁产生的必要条件也就是顺水推舟了：

- 至少存在两个并发事务
- 每个事务都持有锁资源，但是都不会释放
- 每个事务都在申请新的锁资源
- 事务之间形成了锁资源的循环等待

2.2 工作中遇到的死锁

我们在工作中遇到的死锁，绝大多数都是“唯一键”（列值唯一）引起的。好的，那我们先给 worker 表添加一个唯一性约束吧。执行如下 SQL 语句（执行后可以自行验证下是否添加成功）：

```
mysql> ALTER TABLE `worker` ADD UNIQUE(`name`);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

在现实场景中，不同的 worker 有相同的 name 当然是正常的。但是，如果业务并发量比较大，相同的 name 反复插入，不仅会出现 Unique Key 冲突，还可能会出现死锁。为了模拟这一类死锁，我们需要开启三个 MySQL 客户端（确实有点多，一定不要搞乱了），称为：“会话 A”、“会话 B”、“会话 C”。下面，开启死锁之旅吧。

```
-- “会话 A” 开启事务（需要关闭自动提交）
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

-- “会话 A” 插入一条 name = Java 的记录（其他字段任意即可）
mysql> INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('A', 'Java', 1800, 0);
Query OK, 1 row affected (0.00 sec)

-- “会话 B” 开启事务（需要关闭自动提交）
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

-- “会话 B” 插入同样的 Java 记录，事务卡住（如果试验过程中出现了锁等待超时，重新执行插入即可）
mysql> INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('A', 'Java', 1800, 0);

-- “会话 C” 开启事务（需要关闭自动提交）
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

-- “会话 C” 插入同样的 Java 记录，事务卡住（如果试验过程中出现了锁等待超时，重新执行插入即可）
mysql> INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('A', 'Java', 1800, 0);

-- “会话 A” 回滚
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

-- “会话 B” 插入成功（注意锁等待时间）
mysql> INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('A', 'Java', 1800, 0);
Query OK, 1 row affected (14.12 sec)

-- “会话 C” 死锁了，且被 MySQL KILL 掉了
mysql> INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('A', 'Java', 1800, 0);
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

之所以会出现死锁，是因为在插入数据时，MySQL 会给行记录加上排它锁。示例中的三个操作都开启了事务，其中一个（“会话 A”）获取了排它锁开始插入，之后的事务（“会话 B”，“会话 C”）再去执行时会出现 Duplicate Key（重复的值）问题，此时它们都会去申请该行记录的共享锁。如果这个时候，占据排它锁的事务出现回滚（“会话 A”），另外的两个事务会同时去申请排它锁。但是，在数据库中，排它锁和共享锁是互斥资源，也就导致了死锁。

之所以在出现 Duplicate Key 时会加上共享锁，是因为冲突检测是读操作，所以，冲突之后的轮询仍然会有共享限制。我们在工作中遇到的死锁几乎都是由这类情况引起的，那么，参照当前的案例，你能再列举几个工作中死锁的场景吗？

3 怎样发现系统中的死锁

死锁问题并不容易解决，但是，首先第一步，你需要知道哪里发生了死锁。在 MySQL 中，我们可以通过查看命令输出和系统表数据来定位死锁问题，下面，一起来看看吧。

3.1 命令输出锁信息

MySQL 提供了三个常用的系统命令，用于查看会话状态、锁信息以及死锁记录信息。首先，查看会话状态可以使用 PROCESSLIST 命令（需要有 root 权限，之前已经见到过了），如下所示。

```
mysql> SHOW FULL PROCESSLIST;
+----+------+-----------------+--------------------+---------+------+----------+-----------------------+
| Id | User | Host            | db                 | Command | Time | State    | Info                  |
+----+------+-----------------+--------------------+---------+------+----------+-----------------------+
|  3 | root | localhost:50112 | imooc_mysql        | Sleep   |   54 |          | NULL                  |
|  4 | root | localhost:50113 | NULL               | Sleep   |   72 |          | NULL                  |
|  5 | root | localhost       | imooc_mysql        | Sleep   |  725 |          | NULL                  |
|  6 | root | localhost       | imooc_mysql        | Sleep   |  934 |          | NULL                  |
|  7 | root | localhost       | imooc_mysql        | Query   |    0 | starting | SHOW FULL PROCESSLIST |
|  8 | root | localhost:52292 | information_schema | Sleep   |   62 |          | NULL                  |
|  9 | root | localhost:52293 | NULL               | Sleep   |   32 |          | NULL                  |
| 10 | root | localhost       | imooc_mysql        | Sleep   |  946 |          | NULL                  |
+----+------+-----------------+--------------------+---------+------+----------+-----------------------+
```

我们需要重点关注 State（会话状态）字段，如果有很多会话的 State 字段值是 `waiting for ... lock`，基本可以判断当前系统中出现了死锁。但是，这个命令只能告诉我们这么多，具体是哪张表、哪条 SQL 语句引起的死锁，我们还是一无所知的。

MySQL 提供了一个命令，可以用来查询是否锁表。在具体的介绍它之前，我们先来看一看它的使用方法（我觉得倒叙的方式会更好的帮助你理解）：

```
-- 先去显示的锁住 imooc_mysql 库中的 worker 表（可以锁住任意表）
mysql> LOCK TABLES imooc_mysql.worker READ;
Query OK, 0 rows affected (0.00 sec)

-- 查询锁表的情况
mysql> SHOW OPEN TABLES WHERE In_use > 0;
+-------------+--------+--------+-------------+
| Database    | Table  | In_use | Name_locked |
+-------------+--------+--------+-------------+
| imooc_mysql | worker |      1 |           0 |
+-------------+--------+--------+-------------+
1 row in set (0.00 sec)

-- 操作完了之后，别忘了释放锁
mysql> UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

从以上操作过程可以看到，我们先去锁住 worker 表，并通过 `SHOW OPEN TABLES` 命令确认了这一情况，最后释放了锁。`SHOW OPEN TABLES` 的作用是列出当前在表缓存中打开的非临时表，语法如下：

```
SHOW OPEN TABLES
    [{FROM | IN} db_name]
    [LIKE 'pattern' | WHERE expr]
```

其输出包含四列，含义分别是：

- Database：库名
- Table：表名
- In_use：表锁或锁请求的数量
- Name_locked：是否锁定表名（删除表或重命名表时需要）

我们可以使用它来查看当前系统中被锁定的表，以及判断某一张表是否被锁定。例如：我想看一看 worker 表是否被锁定，可以这样（注意语法）：

```
-- In_use 字段为 0，代表 worker 表没有被锁定
mysql> SHOW OPEN TABLES FROM imooc_mysql like 'worker';
+-------------+--------+--------+-------------+
| Database    | Table  | In_use | Name_locked |
+-------------+--------+--------+-------------+
| imooc_mysql | worker |      0 |           0 |
+-------------+--------+--------+-------------+
1 row in set (0.00 sec)
```

在讲解第三个系统命令（用于查看死锁记录信息）之前，我们先来说一说 InnoDB 的监控机制。MySQL 提供了一套 InnoDB 的监控机制，主要分为两种：标准监控和锁监控。想要获取死锁日志，我们需要开启 InnoDB 的标准监控，但是通常锁监控最好也打开，它可以提供一些额外的锁信息。打开方式如下：

```
-- 开启标准监控
SET GLOBAL innodb_status_output = ON;

-- 关闭标准监控
SET GLOBAL innodb_status_output = OFF;

-- 开启锁监控
SET GLOBAL innodb_status_output_locks = ON;

-- 关闭锁监控
SET GLOBAL innodb_status_output_locks = OFF;
```

执行命令 `SHOW ENGINE INNODB STATUS` 可以查看死锁记录信息，但是它有个限制，只能拿到最近一次的死锁日志（这也基本上够用了，因为面对每一个可能发生的死锁，我们都应该去极力避免或解决）。命令打印信息及日志分析如下所示（内容太多了，部分信息使用省略号代替）：

```
mysql> SHOW ENGINE INNODB STATUS\G
.....
-- 这里是一个事务
*** (1) TRANSACTION:
-- ACTIVE 7 sec 表示事务活动时间，inserting 为事务当前正在运行的状态，可能的事务状态有：fetching rows，updating，deleting，inserting 等等
TRANSACTION 8026, ACTIVE 7 sec inserting
-- tables in use 1 表示有一个表被使用，locked 1 表示有一个表锁
mysql tables in use 1, locked 1
-- LOCK WAIT 表示事务正在等待锁
LOCK WAIT 4 lock struct(s), heap size 1136, 2 row lock(s), undo log entries 1
-- 事务的线程信息，以及数据库 IP 地址和数据库名
MySQL thread id 5, OS thread handle 123145495121920, query id 3777 localhost root update
-- 这里显示的是正在等待锁的 SQL 语句，死锁日志里每个事务都只显示一条 SQL 语句
INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('A', 'Java', 1800, 0)
-- 这里显示的是事务正在等待什么锁，RECORD LOCKS 表示记录锁
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 43 page no 6 n bits 80 index name of table `imooc_mysql`.`worker` trx id 8026 lock_mode X locks gap before rec insert inte
ntion waiting
Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
.....

-- 这里是第二个事务，与第一个事务的信息基本相同，那么，相同的部分将不再赘述
*** (2) TRANSACTION:
TRANSACTION 8027, ACTIVE 4 sec inserting
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1136, 2 row lock(s), undo log entries 1
MySQL thread id 10, OS thread handle 123145497071616, query id 3779 localhost root update
INSERT INTO `worker` (`type`, `name`, `salary`, `version`) VALUES ('A', 'Java', 1800, 0)
-- 标识事务二持有什么锁，这个锁往往就是事务一处于锁等待的原因
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 43 page no 6 n bits 80 index name of table `imooc_mysql`.`worker` trx id 8027 lock mode S locks gap before rec
Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
.....

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 43 page no 6 n bits 80 index name of table `imooc_mysql`.`worker` trx id 8027 lock_mode X locks gap before rec insert inte
ntion waiting
Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
.....
```

从这个命令的输出内容中可以看到大量的死锁日志信息，但是仅仅凭借这些日志还是很难定位死锁的，只是知道个大概（可能是执行了哪些语句触发了死锁）。也就是说，想要确定死锁，除了通过系统命令的输出之外，还应该去结合应用程序的代码来进行分析。

3.2 InnoDB 引擎关于锁的表

MySQL5.5 之后，information_schema 系统库中增加了三张关于锁的表（注意，是与 InnoDB 相关的）：

- INNODB_TRX：当前运行的事务
- INNODB_LOCKS：当前锁定的事务
- INNODB_LOCK_WAITS：当前等待的事务

下面，我们来依次解读下这三张表。首先，查询一下 `INFORMATION_SCHEMA.INNODB_TRX` 表：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX\G
*************************** 1. row ***************************
                    trx_id: 8011  -- 事务 ID
                 trx_state: RUNNING -- 事务状态
               trx_started: 2019-12-03 13:09:11 -- 事务开始时间
     trx_requested_lock_id: NULL  -- 等待事务的锁 ID
          trx_wait_started: NULL  -- 事务开始等待的时间
                trx_weight: 2 -- 事务的权重，反映了一个事务修改和锁住的行数
       trx_mysql_thread_id: 6 -- 事务线程 id
                 trx_query: NULL  -- 事务 SQL 语句
       trx_operation_state: NULL  -- 事务当前运行状态
         trx_tables_in_use: 0 -- 事务中有多少个表被使用
         trx_tables_locked: 1 -- 事务中有多少个表被锁住
          trx_lock_structs: 1 -- 事务保留的锁数量
     trx_lock_memory_bytes: 1136  -- 事务锁住的内存大小，单位为 BYTES
           trx_rows_locked: 0 -- 事务锁住的行数
         trx_rows_modified: 1 -- 事务更改的行数
   trx_concurrency_tickets: 0 -- 事务并发数
       trx_isolation_level: REPEATABLE READ -- 事务隔离级别
         trx_unique_checks: 1 -- 是否打开唯一性检查的标识
    trx_foreign_key_checks: 1 -- 是否打开外键检查的标识
trx_last_foreign_key_error: NULL  -- 事务最后一次外键错误信息
 trx_adaptive_hash_latched: 0 -- 自适应散列索引是否被当前事务锁住的标识
 trx_adaptive_hash_timeout: 0 -- 是否立刻放弃为自适应散列索引搜索 LATCH 的标识
          trx_is_read_only: 0 -- 事务是否是只读的
trx_autocommit_non_locking: 0 -- 事务的自动提交是否被打开
1 row in set (0.00 sec)
```

INNODB_TRX 表记录了当前处于运行状态的所有事务，包含非常详细的信息，例如：事务是否正在等待一个锁、事务是否正在执行等等。各个列表达的含义已经在查询 SQL 中给出，弄懂了每一列的含义，也就基本上明白了这张表的含义。

相对于复杂的 INNODB_TRX，INNODB_LOCKS 表就简单许多了。我们来查询下看看吧：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
+--------------+-------------+-----------+-----------+------------------------+------------+------------+-----------+----------+-----------+
| lock_id      | lock_trx_id | lock_mode | lock_type | lock_table             | lock_index | lock_space | lock_page | lock_rec | lock_data |
+--------------+-------------+-----------+-----------+------------------------+------------+------------+-----------+----------+-----------+
| 8012:43:6:12 | 8012        | S         | RECORD    | `imooc_mysql`.`worker` | name       |         43 |         6 |       12 | 'Java'    |
| 8011:43:6:12 | 8011        | X         | RECORD    | `imooc_mysql`.`worker` | name       |         43 |         6 |       12 | 'Java'    |
+--------------+-------------+-----------+-----------+------------------------+------------+------------+-----------+----------+-----------+
2 rows in set, 1 warning (0.00 sec)
```

INNODB_LOCKS 记录的是 InnoDB 事务去请求但没有获取到的锁信息和事务阻塞其他事务的锁信息，各个字段的含义也比较简单，解读如下：

- lock_id：锁 ID
- lock_trx_id：占据锁的事务 ID
- lock_mode：锁模式。可取的值包含：S, X, IS, IX, GAP, AUTO_INC, UNKNOWN
- lock_type：锁类型。RECORD 是行锁，TABLE 是表锁
- lock_table：被锁的表名
- lock_index：lock_type 为行锁时，该值为索引名，否则为空
- lock_space：lock_type 为行锁时，该值为锁记录的表空间的 id，否则为空
- lock_page：lock_type 为行锁时，该值为锁记录页数量，否则为空
- lock_rec：lock_type 为行锁时，该值为页内锁记录的堆数，否则为空
- lock_data：被锁的数据

好的，只剩下最后一张表了，恰巧，它也是最简单的，只包含四个数据列。我们 SELECT 一下吧：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS;
+-------------------+-------------------+-----------------+------------------+
| requesting_trx_id | requested_lock_id | blocking_trx_id | blocking_lock_id |
+-------------------+-------------------+-----------------+------------------+
| 8012              | 8012:43:6:12      | 8011            | 8011:43:6:12     |
+-------------------+-------------------+-----------------+------------------+
1 row in set, 1 warning (0.00 sec)
```

这张表记录了事务的锁等待状态。当事务量比较少，我们可以直观的查看，当事务量非常大，锁等待也时常发生的情况下，这个时候就可以通过 INNODB_LOCK_WAITS 表来更加直观的反映出当前的锁等待情况。好吧，同样看看它的每一列是怎样的含义：

- requesting_trx_id：申请锁资源的事务 id
- requested_lock_id：申请的锁的 id
- blocking_trx_id：阻塞的事务 id
- blocking_lock_id：阻塞的锁的 id

关于 InnoDB 引擎这几张有关锁的表，它们更多的是用来查看 MySQL 系统当前的状态。如果想要去定位死锁的原因，更靠谱的做法肯定还是分析死锁日志。

4 关于死锁问题的建议

理论上说，并发度越高，死锁发生的概率就会越大。虽然不一定能做到完全避免死锁，但是，我们仍可以通过一些技巧或优化降低死锁出现的概率。下面，我给出一些开发建议：

- 尽量避免并发修改数据表数据。这里并不是说不允许并发的出现，而是说将并发修改的过程从数据库中移除，例如只在内存中操作高并发数据（可以考虑 Redis）
- 要求每一个事务将需要用到的数据一次性加锁，否则，不允许执行（实现难度太大，且会降低应用的并发度）
- 避免大事务，尽量将大事务拆分为多个小事务去处理（大事务通常占用资源多，耗时长）
- 设置锁等待超时参数：innodb_lock_wait_timeout。并发较高的情况下，大量事务无法获得锁而挂起，会严重的影响系统性能，减少锁等待时间，不做无意义的等待

当然，以上这些只是建议，不一定需要这样做，甚至有些场景下是不妥的。定位死锁是很难的，不仅需要非常了解业务需求，还需要懂得 InnoDB 中的各种锁机制。所以，尽量在早期做好避免死锁的准备工作。

5 总结

不可否认，关于死锁的话题肯定是不简单的。定位死锁与解决死锁都需要非常丰富的经验，所以，不必要担心它的学习难度，也不要吝啬你的学习时间。其实，又何止是死锁呢，对于任何知识点，都是欲速则不达的。经验主义教会我们，你见的多了，自然也就会了。

6 问题

你在工作中遇到过死锁吗？能举例说明吗？

学会模拟死锁，理解其原理的同时，尝试去分析死锁日志？

你在工作中是怎样避免死锁的呢？出现了死锁，又是怎样解决的呢？

7 参考资料

《高性能 MySQL（第三版）》

MySQL 官方文档：InnoDB INFORMATION_SCHEMA Transaction and Locking Information

MySQL 官方文档：Deadlocks in InnoDB

MySQL 官方文档：Configuring Thread Concurrency for InnoDB

MySQL 官方文档：InnoDB Startup Options and System Variables

}

11 面试常见的高级查询 - 连接、联合、子查询

13 学会对MySQL做基准测试，掌握数据库性能