锁

标签： MySQL是怎样运行的

解决并发事务带来问题的两种基本方式

上一章唠叨了事务并发执行时可能带来的各种问题，并发事务访问相同记录的情况大致可以划分为3种：

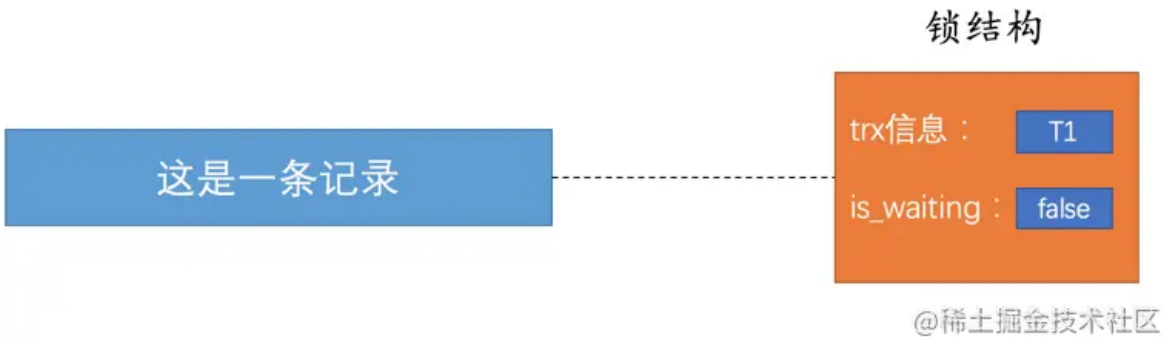
- `读-读`情况：即并发事务相继读取相同的记录。

  读取操作本身不会对记录有一毛钱影响，并不会引起什么问题，所以允许这种情况的发生。

- 写-写情况：即并发事务相继对相同的记录做出改动。

  我们前边说过，在这种情况下会发生脏写的问题，任何一种隔离级别都不允许这种问题的发生。所以在多个未提交事务相继对一条记录做改动时，需要让它们排队执行，这个排队的过程其实是通过锁来实现的。这个所谓的锁其实是一个内存中的结构，在事务执行前本来是没有锁的，也就是说一开始是没有锁结构和记录进行关联的，如图所示：

  当一个事务想对这条记录做改动时，首先会看看内存中有没有与这条记录关联的锁结构，当没有的时候就会在内存中生成一个锁结构与之关联。比方说事务T1要对这条记录做改动，就需要生成一个锁结构与之关联：

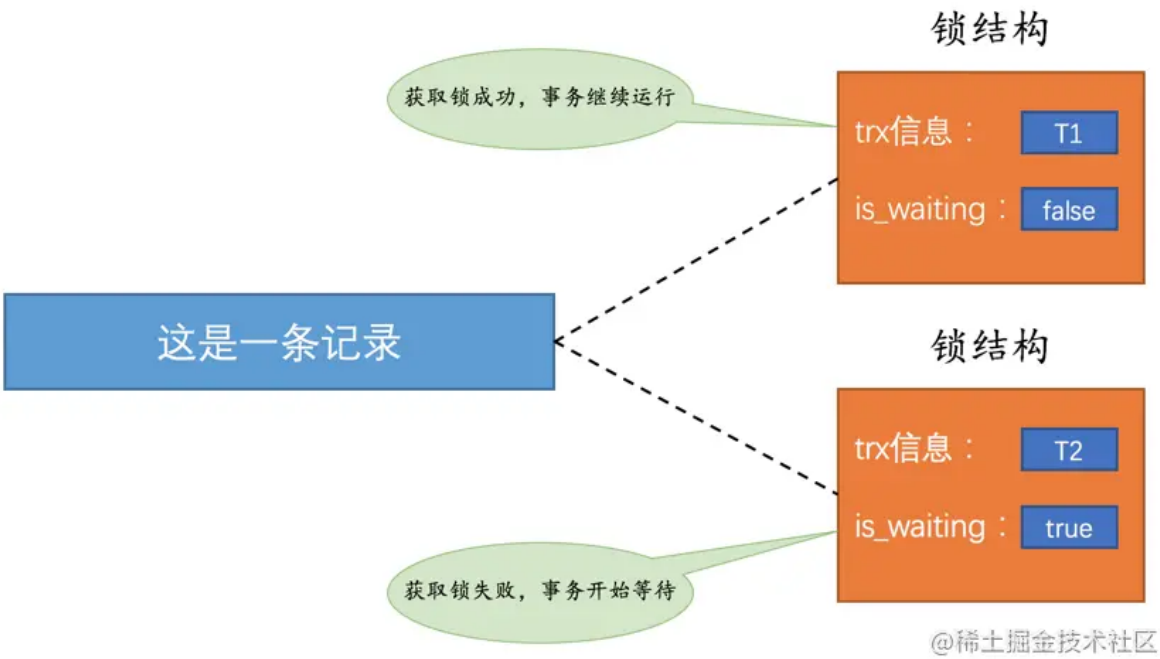


  其实在锁结构里有很多信息，不过为了简化理解，我们现在只把两个比较重要的属性拿了出来：

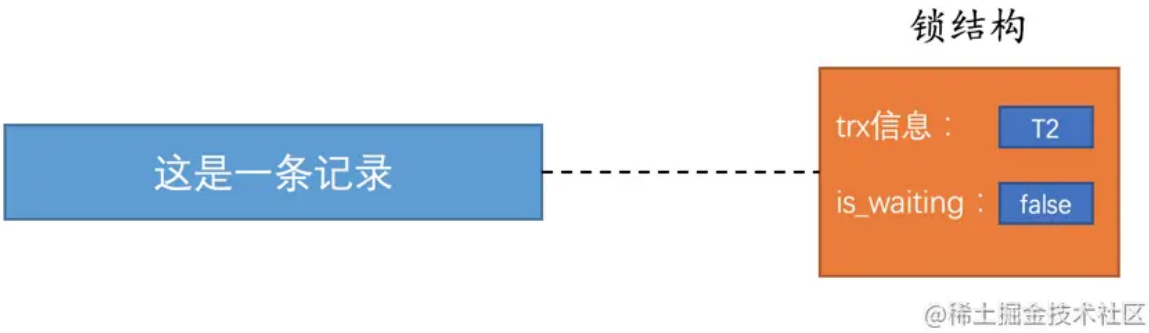
  - trx信息：代表这个锁结构是哪个事务生成的。
  - is_waiting：代表当前事务是否在等待。

  如图所示，当事务T1改动了这条记录后，就生成了一个锁结构与该记录关联，因为之前没有别的事务为这条记录加锁，所以is_waiting属性就是false，我们把这个场景就称之为获取锁成功，或者加锁成功，然后就可以继续执行操作了。

在事务 `T1` 提交之前，另一个事务 `T2` 也想对该记录做改动，那么先去看看有没有 `锁结构` 与这条记录关联，发现有一个 `锁结构` 与之关联后，然后也生成了一个 `锁结构` 与这条记录关联，不过 `锁结构` 的 `is_waiting` 属性值为 `true` ，表示当前事务需要等待，我们把这个场景就称之为获取锁失败，或者加锁失败，或者没有成功的获取到锁，画个图表示就是这样：



在事务 `T1` 提交之后，就会把该事务生成的 `锁结构` 释放掉，然后看看还有没有别的事务在等待获取锁，发现了事务 `T2` 还在等待获取锁，所以把事务 `T2` 对应的锁结构的 `is_waiting` 属性设置为 `false` ，然后把该事务对应的线程唤醒，让它继续执行，此时事务 `T2` 就算获取到锁了。效果图就是这样：



我们总结一下后续内容中可能用到的几种说法，以免大家混淆：

- 不加锁

  意思就是不需要在内存中生成对应的 `锁结构` ，可以直接执行操作。

- 获取锁成功，或者加锁成功

  意思就是在内存中生成了对应的 `锁结构` ，而且锁结构的 `is_waiting` 属性为 `false` ，也就是事务可以继续执行操作。

- 获取锁失败，或者加锁失败，或者没有获取到锁

  意思就是在内存中生成了对应的`锁结构`，不过锁结构的`is_waiting`属性为`true`，也就是事务需要等待，不可以继续执行操作。

> 小贴士： 这里只是对锁结构做了一个非常简单的描述，我们后边会详细唠叨唠叨锁结构的，稍安勿躁。

- 读-写 或 写-读 情况：也就是一个事务进行读取操作，另一个进行改动操作。

  我们前边说过，这种情况下可能发生 脏读 、 不可重复读 、 幻读 的问题。

  > 小贴士： 幻读问题的产生是因为某个事务读了一个范围的记录，之后别的事务在该范围内插入了新记录，该事务再次读取该范围的记录时，可以读到新插入的记录，所以幻读问题准确的说并不是因为读取和写入一条相同记录而产生的，这一点要注意一下。

  SQL标准 规定不同隔离级别下可能发生的问题不一样：

  - 在 READ UNCOMMITTED 隔离级别下， 脏读 、 不可重复读 、 幻读 都可能发生。
  - 在 READ COMMITTED 隔离级别下， 不可重复读 、 幻读 可能发生， 脏读 不可以发生。
  - 在 REPEATABLE READ 隔离级别下， 幻读 可能发生， 脏读 和 不可重复读 不可以发生。
  - 在 SERIALIZABLE 隔离级别下，上述问题都不可以发生。

  不过各个数据库厂商对 SQL标准 的支持都可能不一样，与 SQL标准 不同的一点就是， MySQL 在 REPEATABLE READ 隔离级别实际上就已经解决了 幻读 问题。

  怎么解决 脏读 、 不可重复读 、 幻读 这些问题呢？其实有两种可选的解决方案：

  - 方案一：读操作利用多版本并发控制（ MVCC ），写操作进行 加锁 。

    所谓的 MVCC 我们在前一章有过详细的描述，就是通过生成一个 ReadView ，然后通过 ReadView 找到符合条件的记录版本（历史版本是由 undo日志 构建的），其实就像是在生成 ReadView 的那个时刻做了一次时间静止（就像用相机拍了一个快照），查询语句只能读到在生成 ReadView 之前已提交事务所做的更改，在生成 ReadView 之前未提交的事务或者之后才开启的事务所做的更改是看不到的。而写操作肯定针对的是最新版本的记录，读记录的历史版本和改动记录的最新版本本身并不冲突，也就是采用 MVCC 时， 读-写 操作并不冲突。

    > 小贴士： 我们说过普通的SELECT语句在READ COMMITTED和REPEATABLE READ隔离级别下会使用到MVCC读取记录。在READ COMMITTED隔离级别下，一个事务在执行过程中每次执行SELECT操作时都会生成一个ReadView，ReadView的存在本身就保证了事务不可以读取到未提交的事务所做的更改，也就是避免了脏读现象；REPEATABLE READ隔离级别下，一个事务在执行过程中只有第一次执行SELECT操作才会生成一个ReadView，之后的SELECT操作都复用这个ReadView，这样也就避免了不可重复读和幻读的问题。

- 方案二：读、写操作都采用 `加锁` 的方式。

  如果我们的一些业务场景不允许读取记录的旧版本，而是每次都必须去读取记录的最新版本，比方在银行存款的事务中，你需要先把账户的余额读出来，然后将其加上本次存款的数额，最后再写到数据库中。在将账户余额读取出来后，就不想让别的事务再访问该余额，直到本次存款事务执行完成，其他事务才可以访问账户的余额。这样在读取记录的时候也就需要对其进行 `加锁` 操作，这样也就意味着 `读` 操作和 `写` 操作也像 `写-写` 操作那样排队执行。

  > 小贴士： 我们说脏读的产生是因为当前事务读取了另一个未提交事务写的一条记录，如果另一个事务在写记录的时候就给这条记录加锁，那么当前事务就无法继续读取该记录了，所以也就不会有脏读问题的产生了。不可重复读的产生是因为当前事务先读取一条记录，另外一个事务对该记录做了改动之后并提交之后，当前事务再次读取时会获得不同的值，如果在当前事务读取记录时就给该记录加锁，那么另一个事务就无法修改该记录，自然也不会发生不可重复读了。我们说幻读问题的产生是因为当前事务读取了一个范围的记录，然后另外的事务向该范围内插入了新记录，当前事务再次读取该范围的记录时发现了新插入的新记录，我们把新插入的那些记录称之为幻影记录。采用加锁的方式解决幻读问题就有那么一丢丢麻烦了，因为当前事务在第一次读取记录时那些幻影记录并不存在，所以读取的时候加锁就有点尴尬 —— 因为你并不知道给谁加锁，没关系，这难不倒设计InnoDB的大叔的，我们稍后揭晓答案，稍安勿躁。

很明显，采用 `MVCC` 方式的话， `读-写` 操作彼此并不冲突，性能更高，采用 `加锁` 方式的话， `读-写` 操作彼此需要排队执行，影响性能。一般情况下我们当然愿意采用 `MVCC` 来解决 `读-写` 操作并发执行的问题，但是业务在某些特殊情况下，要求必须采用 `加锁` 的方式执行，那也是没有办法的事。

一致性读（Consistent Reads）

事务利用 `MVCC` 进行的读取操作称之为 `一致性读` ，或者 `一致性无锁读` ，有的地方也称之为 `快照读` 。所有普通的 `SELECT` 语句（ `plain SELECT` ）在 `READ COMMITTED` 、 `REPEATABLE READ` 隔离级别下都算是 `一致性读` ，比方说：

```
SELECT * FROM t;
SELECT * FROM t1 INNER JOIN t2 ON t1.col1 = t2.col2
```

`一致性读` 并不会对表中的任何记录做 `加锁` 操作，其他事务可以自由的对表中的记录做改动。

锁定读（Locking Reads）

共享锁和独占锁

我们前边说过，并发事务的 `读-读` 情况并不会引起什么问题，不过对于 `写-写` 、 `读-写` 或 `写-读` 这些情况可能会引起一些问题，需要使用 `MVCC` 或者 `加锁` 的方式来解决它们。在使用 `加锁` 的方式解决问题时，由于既要允许 `读-读` 情况不受影响，又要使 `写-写` 、 `读-写` 或 `写-读` 情况中的操作相互阻塞，所以设计 `MySQL` 的大叔给锁分了个类：

- 共享锁，英文名：Shared Locks，简称S锁。在事务要读取一条记录时，需要先获取该记录的S锁。
- 独占锁，也常称排他锁，英文名：Exclusive Locks，简称X锁。在事务要改动一条记录时，需要先获取该记录的X锁。

假如事务T1首先获取了一条记录的S锁之后，事务T2接着也要访问这条记录：

- 如果事务T2想要再获取一个记录的S锁，那么事务T2也会获得该锁，也就意味着事务T1和T2在该记录上同时持有S锁。
- 如果事务T2想要再获取一个记录的X锁，那么此操作会被阻塞，直到事务T1提交之后将S锁释放掉。

如果事务T1首先获取了一条记录的X锁之后，那么不管事务T2接着想获取该记录的S锁还是X锁都会被阻塞，直到事务T1提交。

所以我们说S锁和S锁是兼容的，S锁和X锁是不兼容的，X锁和X锁也是不兼容的，画个表表示一下就是这样：

兼容性	X	S
X	不兼容	不兼容
S	不兼容	兼容

锁定读的语句

我们前边说在采用加锁方式解决脏读、不可重复读、幻读这些问题时，读取一条记录时需要获取一下该记录的S锁，其实这是不严谨的，有时候想在读取记录时就获取记录的X锁，来禁止别的事务读写该记录，为此设计MySQL的大叔提出了两种比较特殊的SELECT语句格式：

- 对读取的记录加S锁：

  ```
  SELECT ... LOCK IN SHARE MODE;
  ```

  也就是在普通的SELECT语句后边加LOCK IN SHARE MODE，如果当前事务执行了该语句，那么它会为读取到的记录加S锁，这样允许别的事务继续获取这些记录的S锁（比方说别的事务也使用SELECT ... LOCK IN SHARE MODE语句来读取这些记录），但是不能获取这些记录的X锁（比方说使用SELECT ... FOR UPDATE语句来读取这些记录，或者直接修改这些记录）。如果别的事务想要获取这些记录的X锁，那么它们会阻塞，直到当前事务提交之后将这些记录上的S锁释放掉。

- 对读取的记录加 `X锁` ：

  ```
  SELECT ... FOR UPDATE;
  ```

  也就是在普通的 `SELECT` 语句后边加 `FOR UPDATE` ，如果当前事务执行了该语句，那么它会为读取到的记录加 `X锁` ，这样既不允许别的事务获取这些记录的 `S锁` （比方说别的事务使用 `SELECT ... LOCK IN SHARE MODE` 语句来读取这些记录），也不允许获取这些记录的 `X锁` （比如说使用 `SELECT ... FOR UPDATE` 语句来读取这些记录，或者直接修改这些记录）。如果别的事务想要获取这些记录的 `S锁` 或者 `X锁` ，那么它们会阻塞，直到当前事务提交之后将这些记录上的 `X锁` 释放掉。

关于更多 `锁定读` 的加锁细节我们稍后会详细唠叨，稍安勿躁。

写操作

平常所用到的 `写操作` 无非是 `DELETE` 、 `UPDATE` 、 `INSERT` 这三种：

- `DELETE` ：

  对一条记录做 `DELETE` 操作的过程其实是先在 `B+` 树中定位到这条记录的位置，然后获取一下这条记录的 `X锁` ，然后再执行 `delete mark` 操作。我们也可以把这个定位待删除记录在 `B+` 树中位置的过程看成是一个获取 `X锁` 的 `锁定读` 。

- `UPDATE` ：

  在对一条记录做 `UPDATE` 操作时分为三种情况：

  - 如果未修改该记录的键值并且被更新的列占用的存储空间在修改前后未发生变化，则先在 `B+` 树中定位到这条记录的位置，然后再获取一下记录的 `X锁` ，最后在原记录的位置进行修改操作。其实我们也可以把这个定位待修改记录在 `B+` 树中位置的过程看成是一个获取 `X锁` 的 `锁定读` 。

  - 如果未修改该记录的键值并且至少有一个被更新的列占用的存储空间在修改前后发生变化，则先在 `B+` 树中定位到这条记录的位置，然后获取一下记录的 `X锁` ，将该记录彻底删除掉（就是把记录彻底移入垃圾链表），最后再插入一条新记录。这个定位待修改记录在 `B+` 树中位置的过程看成是一个获取 `X锁` 的 `锁定读` ，新插入的记录由 `INSERT` 操作提供的 `隐式锁` 进行保护。

  - 如果修改了该记录的键值，则相当于在原记录上做 `DELETE` 操作之后再来一次 `INSERT` 操作，加锁操作就需要按照 `DELETE` 和 `INSERT` 的规则进行了。

- `INSERT` ：

  一般情况下，新插入一条记录的操作并不加锁，设计 `InnoDB` 的大叔通过一种称之为 `隐式锁` 的东东来保护这条新插入的记录在本事务提交前不被别的事务访问，更多细节我们后边看哈～

  > 小贴士： 当然，在一些特殊情况下INSERT操作也是会获取锁的，具体情况我们后边唠叨。

多粒度锁

我们前边提到的`锁`都是针对记录的，也可以被称之为`行级锁`或者`行锁`，对一条记录加锁影响的也只是这条记录而已，我们就说这个锁的粒度比较细；其实一个事务也可以在`表`级别进行加锁，自然就被称之为`表级锁`或者`表锁`，对一个表加锁影响整个表中的记录，我们就说这个锁的粒度比较粗。给表加的锁也可以分为`共享锁`（`S锁`）和`独占锁`（`X锁`）：

- 给表加`S锁`：

  如果一个事务给表加了`S锁`，那么：

  - 别的事务可以继续获得该表的`S锁`
  - 别的事务可以继续获得该表中的某些记录的`S锁`
  - 别的事务不可以继续获得该表的`X锁`
  - 别的事务不可以继续获得该表中的某些记录的`X锁`
- 给表加`X锁`：

  如果一个事务给表加了`X锁`（意味着该事务要独占这个表），那么：

  - 别的事务不可以继续获得该表的`S锁`
  - 别的事务不可以继续获得该表中的某些记录的`S锁`
  - 别的事务不可以继续获得该表的`X锁`
  - 别的事务不可以继续获得该表中的某些记录的`X锁`

上边看着有点啰嗦，为了更好的理解这个表级别的`S锁`和`X锁`，我们举一个现实生活中的例子。不知道各位同学都上过大学没，我们以大学教学楼中的教室为例来分析一下加锁的情况：

- 教室一般都是公用的，我们可以随便选教室进去上自习。当然，教室不是自家的，一间教室可以容纳很多同学同时上自习，每当一个人进去上自习，就相当于在教室门口挂了一把`S锁`，如果很多同学都进去上自习，相当于教室门口挂了很多把`S锁`（类似行级别的`S锁`）。

- 有的时候教室会进行检修，比方说换地板，换天花板，换灯管啥的，这些维修项目并不能同时开展。如果教室针对某个项目进行检修，就不允许别的同学来上自习，也不允许其他维修项目进行，此时相当于教室门口会挂一把`X锁`（类似行级别的`X锁`）。

上边提到的这两种锁都是针对`教室`而言的，不过有时候我们会有一些特殊的需求：

- 有领导要来参观教学楼的环境。

  校领导考虑并不想影响同学们上自习，但是此时不能有教室处于维修状态，所以可以在教学楼门口放置一把 `S锁` （类似表级别的 `S锁` ）。此时：

  - 来上自习的学生们看到教学楼门口有 `S锁` ，可以继续进入教学楼上自习。
  - 修理工看到教学楼门口有 `S锁` ，则先在教学楼门口等着，啥时候领导走了，把教学楼的 `S锁` 撤掉再进入教学楼维修。

- 学校要占用教学楼进行考试。

  此时不允许教学楼中有正在上自习的教室，也不允许对教室进行维修。所以可以在教学楼门口放置一把 `X锁` （类似表级别的 `X锁` ）。此时：

  - 来上自习的学生们看到教学楼门口有 `X锁` ，则需要在教学楼门口等着，啥时候考试结束，把教学楼的 `X锁` 撤掉再进入教学楼上自习。
  - 修理工看到教学楼门口有 `X锁` ，则先在教学楼门口等着，啥时候考试结束，把教学楼的 `X锁` 撤掉再进入教学楼维修。

但是这里头有两个问题：

- 如果我们想对教学楼整体上 `S锁` ，首先需要确保教学楼中的没有正在维修的教室，如果有正在维修的教室，需要等到维修结束才可以对教学楼整体上 `S锁` 。
- 如果我们想对教学楼整体上 `X锁` ，首先需要确保教学楼中的没有上自习的教室以及正在维修的教室，如果有上自习的教室或者正在维修的教室，需要等到全部上自习的同学都上完自习离开，以及维修工维修完教室离开后才可以对教学楼整体上 `X锁` 。

我们在对教学楼整体上锁（ `表锁` ）时，怎么知道教学楼中有没有教室已经被上锁（ `行锁` ）了呢？依次检查每一间教室门口有没有上锁？那这效率也太慢了吧！遍历是不可能遍历的，这辈子也不可能遍历的，于是乎设计 `InnoDB` 的大叔们提出了一种称之为 `意向锁` （英文名： `Intention Locks` ）的东东：

- 意向共享锁，英文名： `Intention Shared Lock` ，简称 `IS锁` 。当事务准备在某条记录上加 `S锁` 时，需要先在表级别加一个 `IS锁` 。
- 意向独占锁，英文名： `Intention Exclusive Lock` ，简称 `IX锁` 。当事务准备在某条记录上加 `X锁` 时，需要先在表级别加一个 `IX锁` 。

视角回到教学楼和教室上来：

- 如果有学生到教室中上自习，那么他先在整栋教学楼门口放一把 `IS锁` （表级锁），然后再到教室门口放一把 `S锁` （行锁）。
- 如果有维修工到教室中维修，那么它先在整栋教学楼门口放一把 `IX锁` （表级锁），然后再到教室门口放一把 `X锁` （行锁）。

之后：

- 如果有领导要参观教学楼，也就是想在教学楼门口前放`S锁`（表锁）时，首先要看一下教学楼门口有没有`IX锁`，如果有，意味着有教室在维修，需要等到维修结束把`IX锁`撤掉后才可以在整栋教学楼上加`S锁`。
- 如果有考试要占用教学楼，也就是想在教学楼门口前放`X锁`（表锁）时，首先要看一下教学楼门口有没有`IS锁`或`IX锁`，如果有，意味着有教室在上自习或者维修，需要等到学生们上完自习以及维修结束把`IS锁`和`IX锁`撤掉后才可以在整栋教学楼上加`X锁`。

> 小贴士：学生在教学楼门口加IS锁时，是不关心教学楼门口是否有IX锁的，维修工在教学楼门口加IX锁时，是不关心教学楼门口是否有IS锁或者其他IX锁的。IS和IX锁只是为了判断当前时间教学楼里有没有被占用的教室用的，也就是在对教学楼加S锁或者X锁时才会用到。

总结一下：IS、IX锁是表级锁，它们的提出仅仅为了在之后加表级别的S锁和X锁时可以快速判断表中的记录是否被上锁，以避免用遍历的方式来查看表中有没有上锁的记录，也就是说其实IS锁和IX锁是兼容的，IX锁和IX锁是兼容的。我们画个表来看一下表级别的各种锁的兼容性：

兼容性	X	IX	S	IS
X	不兼容	不兼容	不兼容	不兼容
IX	不兼容	兼容	不兼容	兼容
S	不兼容	不兼容	兼容	兼容
IS	不兼容	兼容	兼容	兼容

MySQL中的行锁和表锁

上边说的都算是些理论知识，其实`MySQL`支持多种存储引擎，不同存储引擎对锁的支持也是不一样的。当然，我们重点还是讨论`InnoDB`存储引擎中的锁，其他的存储引擎只是稍微提一下～

其他存储引擎中的锁

对于`MyISAM`、`MEMORY`、`MERGE`这些存储引擎来说，它们只支持表级锁，而且这些引擎并不支持事务，所以使用这些存储引擎的锁一般都是针对当前会话来说的。比方说在`Session 1`中对一个表执行`SELECT`操作，就相当于为这个表加了一个表级别的`S锁`，如果在`SELECT`操作未完成时，`Session 2`中对这个表执行`UPDATE`操作，相当于要获取表的`X锁`，此操作会被阻塞，直到`Session 1`中的`SELECT`操作完成，释放掉表级别的`S锁`后，`Session 2`中对这个表执行`UPDATE`操作才能继续获取`X锁`，然后执行具体的更新语句。

> 小贴士： 因为使用MyISAM、MEMORY、MERGE这些存储引擎的表在同一时刻只允许一个会话对表进行写操作，所以这些存储引擎实际上最好用在只读，或者大部分都是读操作，或者单用户的情景下。 另外，在MyISAM存储引擎中有一个称之为Concurrent Inserts的特性，支持在对MyISAM表读取时同时插入记录，这样可以提升一些插入速度。关于更多Concurrent Inserts的细节，我们就不唠叨了，详情可以参考文档。

InnoDB存储引擎中的锁

`InnoDB` 存储引擎既支持表锁，也支持行锁。表锁实现简单，占用资源较少，不过粒度很粗，有时候你仅仅需要锁住几条记录，但使用表锁的话相当于为表中的所有记录都加锁，所以性能比较差。行锁粒度更细，可以实现更精准的并发控制。下边我们详细看一下。

InnoDB中的表级锁

- 表级别的 `S锁` 、 `X锁`

  在对某个表执行 `SELECT` 、 `INSERT` 、 `DELETE` 、 `UPDATE` 语句时， `InnoDB` 存储引擎是不会为这个表添加表级别的 `S锁` 或者 `X锁` 的。

  另外，在对某个表执行一些诸如 `ALTER TABLE` 、 `DROP TABLE` 这类的 `DDL` 语句时，其他事务对这个表并发执行诸如 `SELECT` 、 `INSERT` 、 `DELETE` 、 `UPDATE` 的语句会发生阻塞，同理，某个事务中对某个表执行 `SELECT` 、 `INSERT` 、 `DELETE` 、 `UPDATE` 语句时，在其他会话中对这个表执行 `DDL` 语句也会发生阻塞。这个过程其实是通过在 `server层` 使用一种称之为 `元数据锁` （英文名： `Metadata Locks` ，简称 `MDL` ）东东来实现的，一般情况下也不会使用 `InnoDB` 存储引擎自己提供的表级别的 `S锁` 和 `X锁` 。

  > 小贴士： 在事务简介的章节中我们说过，DDL语句执行时会隐式的提交当前会话中的事务，这主要是DDL语句的执行一般都会在若干个特殊事务中完成，在开启这些特殊事务前，需要将当前会话中的事务提交掉。另外，关于MDL锁并不是我们本章所要讨论的范围，大家可以参阅文档了解哈～

  其实这个 `InnoDB` 存储引擎提供的表级 `S锁` 或者 `X锁` 是相当鸡肋，只会在一些特殊情况下，比方说崩溃恢复过程中用到。不过我们还是可以手动获取一下的，比方说在系统变量 `autocommit=0，innodb_table_locks = 1` 时，手动获取 `InnoDB` 存储引擎提供的表 `t` 的 `S锁` 或者 `X锁` 可以这么写：

  - `LOCK TABLES t READ` ： `InnoDB` 存储引擎会对表 `t` 加表级别的 `S锁` 。
  - `LOCK TABLES t WRITE` ： `InnoDB` 存储引擎会对表 `t` 加表级别的 `X锁` 。

  不过请尽量避免在使用 `InnoDB` 存储引擎的表上使用 `LOCK TABLES` 这样的手动锁表语句，它们并不会提供什么额外的保护，只是会降低并发能力而已。 `InnoDB` 的厉害之处还是实现了更细粒度的行锁，关于表级别的 `S锁` 和 `X锁` 大家了解一下就罢了。

- 表级别的`IS锁`、`IX锁`

  当我们在对使用`InnoDB`存储引擎的表的某些记录加`S锁`之前，那就需要先在表级别加一个`IS锁`，当我们在对使用`InnoDB`存储引擎的表的某些记录加`X锁`之前，那就需要先在表级别加一个`IX锁`。`IS锁`和`IX锁`的使命只是为了后续在加表级别的`S锁`和`X锁`时判断表中是否有已经被加锁的记录，以避免用遍历的方式来查看表中有没有上锁的记录。更多关于`IS锁`和`IX锁`的解释我们上边都唠叨过了，就不赘述了。

- 表级别的 AUTO-INC锁

  在使用 MySQL 过程中，我们可以为表的某个列添加 AUTO_INCREMENT 属性，之后在插入记录时，可以不指定该列的值，系统会自动为它赋上递增的值，比方说我们有一个表：

```
CREATE TABLE t (
    id INT NOT NULL AUTO_INCREMENT,
    c VARCHAR(100),
    PRIMARY KEY (id)
) Engine=InnoDB CHARSET=utf8;
```

  由于这个表的 id 字段声明了 AUTO_INCREMENT ，也就意味着在书写插入语句时不需要为其赋值，比方说这样：

```
INSERT INTO t(c) VALUES('aa'), ('bb');
```

  上边的插入语句并没有为 id 列显式赋值，所以系统会自动为它赋上递增的值，效果就是这样：

```
mysql> SELECT * FROM t;
+----+------+
| id | c    |
+----+------+
|  1 | aa   |
|  2 | bb   |
+----+------+
2 rows in set (0.00 sec)
```

  系统实现这种自动给 AUTO_INCREMENT 修饰的列递增赋值的原理主要是两个：

  - 采用 AUTO-INC 锁，也就是在执行插入语句时就在表级别加一个 AUTO-INC 锁，然后为每条待插入记录的 AUTO_INCREMENT 修饰的列分配递增的值，在该语句执行结束后，再把 AUTO-INC 锁释放掉。这样一个事务在持有 AUTO-INC 锁的过程中，其他事务的插入语句都要被阻塞，可以保证一个语句中分配的递增值是连续的。

    如果我们的插入语句在执行前不可以确定具体要插入多少条记录（无法预计即将插入记录的数量），比方说使用 INSERT ... SELECT 、 REPLACE ... SELECT 或者 LOAD DATA 这种插入语句，一般是使用 AUTO-INC 锁为 AUTO_INCREMENT 修饰的列生成对应的值。

    > 小贴士： 需要注意一下的是，这个AUTO-INC锁的作用范围只是单个插入语句，插入语句执行完成后，这个锁就被释放了，跟我们之前介绍的锁在事务结束时释放是不一样的。

- 采用一个轻量级的锁，在为插入语句生成 `AUTO_INCREMENT` 修饰的列的值时获取一下这个轻量级锁，然后生成本次插入语句需要用到的 `AUTO_INCREMENT` 列的值之后，就把该轻量级锁释放掉，并不需要等到整个插入语句执行完才释放锁。

  如果我们的插入语句在执行前就可以确定具体要插入多少条记录，比方说我们上边举的关于表 `t` 的例子中，在语句执行前就可以确定要插入2条记录，那么一般采用轻量级锁的方式对 `AUTO_INCREMENT` 修饰的列进行赋值。这种方式可以避免锁定表，可以提升插入性能。

> 小贴士： 设计InnoDB的大叔提供了一个称之为innodb_autoinc_lock_mode的系统变量来控制到底使用上述两种方式中的哪种来为AUTO_INCREMENT修饰的列进行赋值，当innodb_autoinc_lock_mode值为0时，一律采用AUTO-INC锁；当innodb_autoinc_lock_mode值为2时，一律采用轻量级锁；当innodb_autoinc_lock_mode值为1时，两种方式混着来（也就是在插入记录数量确定时采用轻量级锁，不确定时使用AUTO-INC锁）。不过当innodb_autoinc_lock_mode值为2时，可能会造成不同事务中的插入语句为AUTO_INCREMENT修饰的列生成的值是交叉的，在有主从复制的场景中是不安全的。

InnoDB中的行级锁

很遗憾的通知大家一个不好的消息，上边讲的都是铺垫，本章真正的重点才刚刚开始[手动偷笑]。

`行锁` ，也称为 `记录锁` ，顾名思义就是在记录上加的锁。不过设计 `InnoDB` 的大叔很有才，一个 `行锁` 玩出了各种花样，也就是把 `行锁` 分成了各种类型。换句话说即使对同一条记录加 `行锁` ，如果类型不同，起到的功效也是不同的。为了故事的顺利发展，我们还是先将之前唠叨 `MVCC` 时用到的表抄一遍：

```
CREATE TABLE hero (
    number INT,
    name VARCHAR(100),
    country varchar(100),
    PRIMARY KEY (number)
) Engine=InnoDB CHARSET=utf8;
```

我们主要是想用这个表存储三国时的英雄，然后向这个表里插入几条记录：

```
INSERT INTO hero VALUES
    (1, 'l刘备', '蜀'),
    (3, 'z诸葛亮', '蜀'),
    (8, 'c曹操', '魏'),
    (15, 'x荀彧', '魏'),
    (20, 's孙权', '吴');
```

现在表里的数据就是这样的：

```
mysql> SELECT * FROM hero;
+--------+------------+---------+
| number | name       | country |
+--------+------------+---------+
|      1 | l刘备      | 蜀      |
|      3 | z诸葛亮    | 蜀      |
|      8 | c曹操      | 魏      |
|     15 | x荀彧      | 魏      |
|     20 | s孙权      | 吴      |
+--------+------------+---------+
5 rows in set (0.01 sec)
```

> 小贴士： 不是说好的存储三国时的英雄么，你在搞什么，为啥要在'刘备'、'曹操'、'孙权'前边加上'l'、'c'、's'这几个字母呀？这个主要是因为我们采用utf8字符集，该字符集并没有对应的按照汉语拼音进行排序的比较规则，也就是说'刘备'、'曹操'、'孙权'这几个字符串的排序并不是按照它们汉语拼音进行排序的，我怕大家懵逼，所以在汉字前边加上了汉字对应的拼音的第一个字母，这样在排序时就是按照汉语拼音进行排序，大家也不懵逼了。 另外，我们故意把各条记录number列的值搞得很分散，后边会用到，稍安勿躁哈～

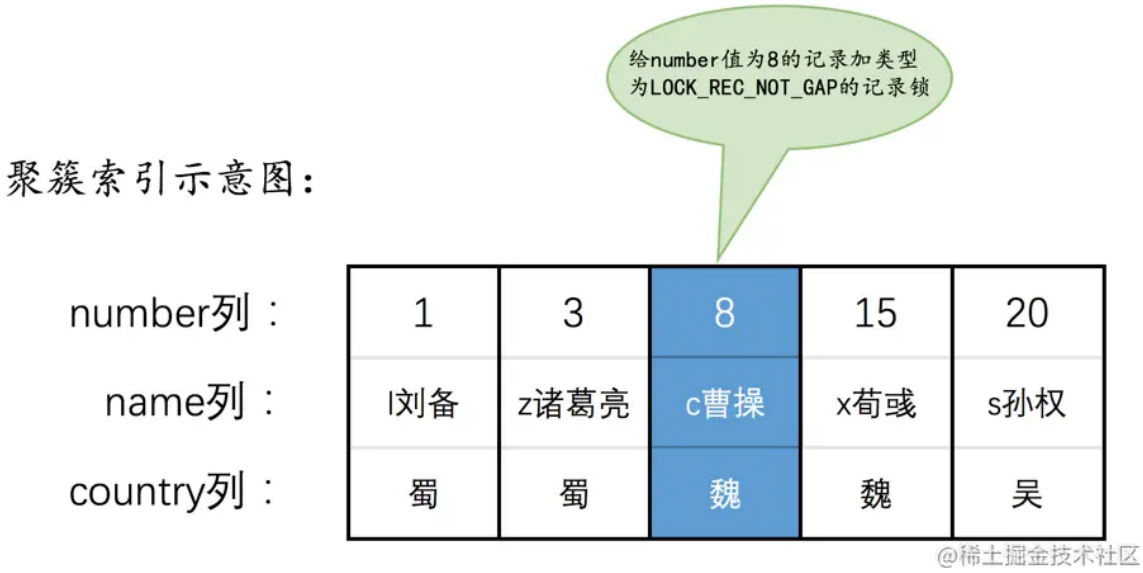
我们把 `hero` 表中的聚簇索引的示意图画一下：

当然，我们把 `B+树` 的索引结构做了一个超级简化，只把索引中的记录给拿了出来，我们这里只是想强调聚簇索引中的记录是按照主键大小排序的，并且省略掉了聚簇索引中的隐藏列，大家心里明白就好（不理解索引结构的同学可以去前边的文章中查看）。

现在准备工作做完了，下边我们来看看都有哪些常用的 `行锁类型`。

- `Record Locks` ：

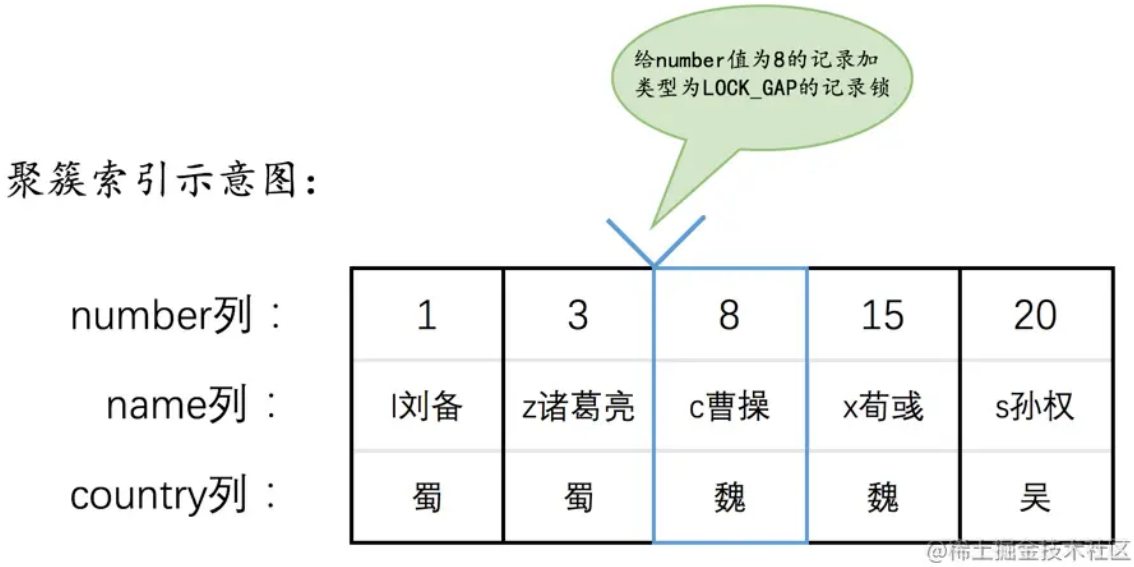
  我们前边提到的记录锁就是这种类型，也就是仅仅把一条记录锁上，我决定给这种类型的锁起一个比较不正经的名字：`正经记录锁` （请允许我皮一下，我实在不知道该叫个啥名好）。官方的类型名称为：`LOCK_REC_NOT_GAP` 。比方说我们把 `number` 值为 `8` 的那条记录加一个 `正经记录锁` 的示意图如下：



  `正经记录锁` 是有 `S锁` 和 `X锁` 之分的，让我们分别称之为 `S型正经记录锁` 和 `X型正经记录锁` 吧（听起来有点怪怪的），当一个事务获取了一条记录的 `S型正经记录锁` 后，其他事务也可以继续获取该记录的 `S型正经记录锁` ，但不可以继续获取 `X型正经记录锁` ；当一个事务获取了一条记录的 `X型正经记录锁` 后，其他事务既不可以继续获取该记录的 `S型正经记录锁` ，也不可以继续获取 `X型正经记录锁` ；

- Gap Locks ：

  我们说 MySQL 在 REPEATABLE READ 隔离级别下是可以解决幻读问题的，解决方案有两种，可以使用 MVCC 方案解决，也可以采用 加锁 方案解决。但是在使用 加锁 方案解决时有个大问题，就是事务在第一次执行读取操作时，那些幻影记录尚不存在，我们无法给这些幻影记录加上 正经记录锁 。不过这难不倒设计 InnoDB 的大叔，他们提出了一种称之为 Gap Locks 的锁，官方的类型名称为： LOCK_GAP ，我们也可以简称为 gap锁 。比方说我们把 number 值为 8 的那条记录加一个 gap锁 的示意图如下：



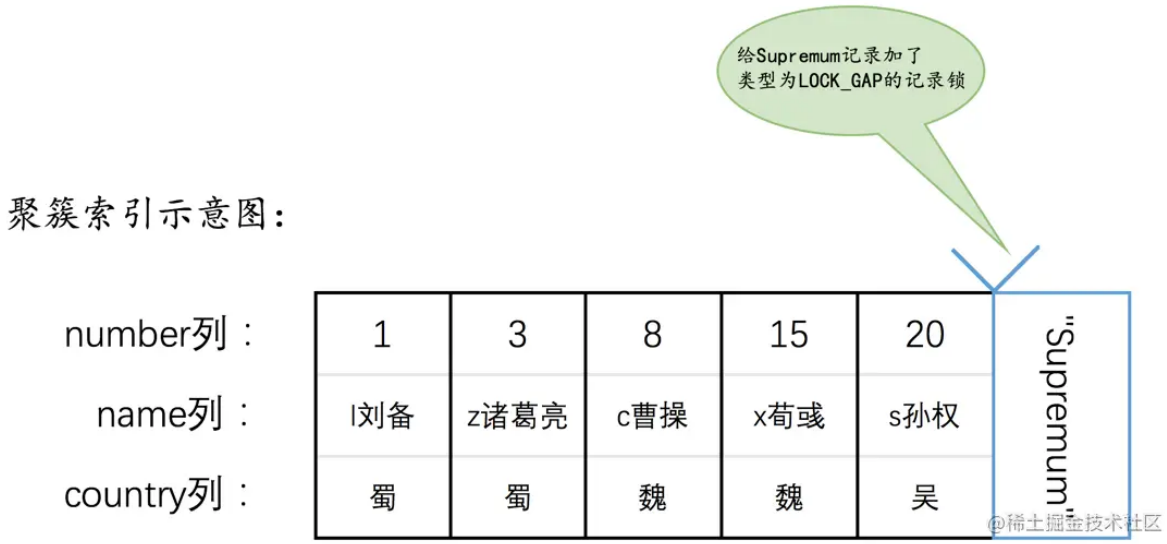
  如图中为 number 值为 8 的记录加了 gap锁 ，意味着不允许别的事务在 number 值为 8 的记录前边的 间隙 插入新记录，其实就是 number 列的值 (3, 8) 这个区间的新记录是不允许立即插入的。比方说有另外一个事务再想插入一条 number 值为 4 的新记录，它定位到该条新记录的下一条记录的 number 值为8，而这条记录上又有一个 gap锁 ，所以就会阻塞插入操作，直到拥有这个 gap锁 的事务提交了之后， number 列的值在区间 (3, 8) 中的新记录才可以被插入。

  这个 gap锁 的提出仅仅是为了防止插入幻影记录而提出的，虽然有 共享gap锁 和 独占gap锁 这样的说法，但是它们起到的作用都是相同的。而且如果你对一条记录加了 gap锁 （不论是 共享gap锁 还是 独占gap锁 ），并不会限制其他事务对这条记录加 正经记录锁 或者继续加 gap锁 ，再强调一遍， gap锁 的作用仅仅是为了防止插入幻影记录的而已。

  不知道大家发现了一个问题没，给一条记录加了 gap锁 只是不允许其他事务往这条记录前边的间隙插入新记录，那对于最后一条记录之后的间隙，也就是 hero 表中 number 值为 20 的记录之后的间隙该咋办呢？也就是说给哪条记录加 gap锁 才能阻止其他事务插入 number 值在 (20, +∞) 这个区间的新记录呢？这时候应该想起我们在前边唠叨 数据页 时介绍的两条伪记录了：

  - Infimum 记录，表示该页面中最小的记录。
  - Supremum 记录，表示该页面中最大的记录。

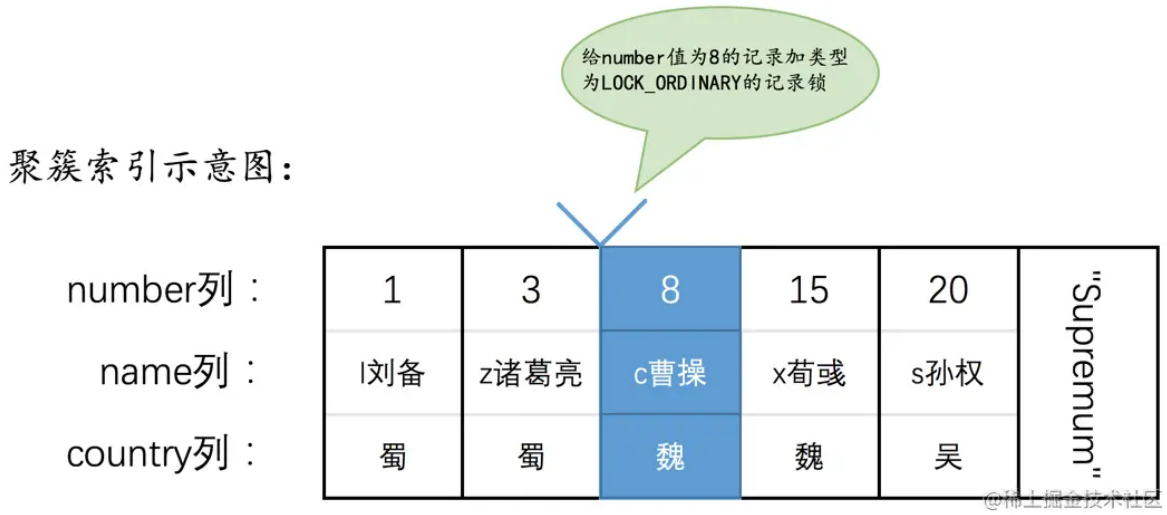
为了实现阻止其他事务插入 `number` 值在 `(20, +∞)` 这个区间的新记录，我们可以给索引中的最后一条记录，也就是 `number` 值为 `20` 的那条记录所在页面的 `Supremum` 记录加上一个 `gap锁` ，画个图就是这样：



这样就可以阻止其他事务插入 `number` 值在 `(20, +∞)` 这个区间的新记录。为了大家理解方便，之后的索引示意图中都会把这个 `Supremum` 记录画出来。

- `Next-Key Locks` ：

有时候我们既想锁住某条记录，又想阻止其他事务在该记录前边的 `间隙` 插入新记录，所以设计 `InnoDB` 的大叔们就提出了一种称之为 `Next-Key Locks` 的锁，官方的类型名称为： `LOCK_ORDINARY` ，我们也可以简称为 `next-key锁` 。比方说我们把 `number` 值为 `8` 的那条记录加一个 `next-key锁` 的示意图如下：

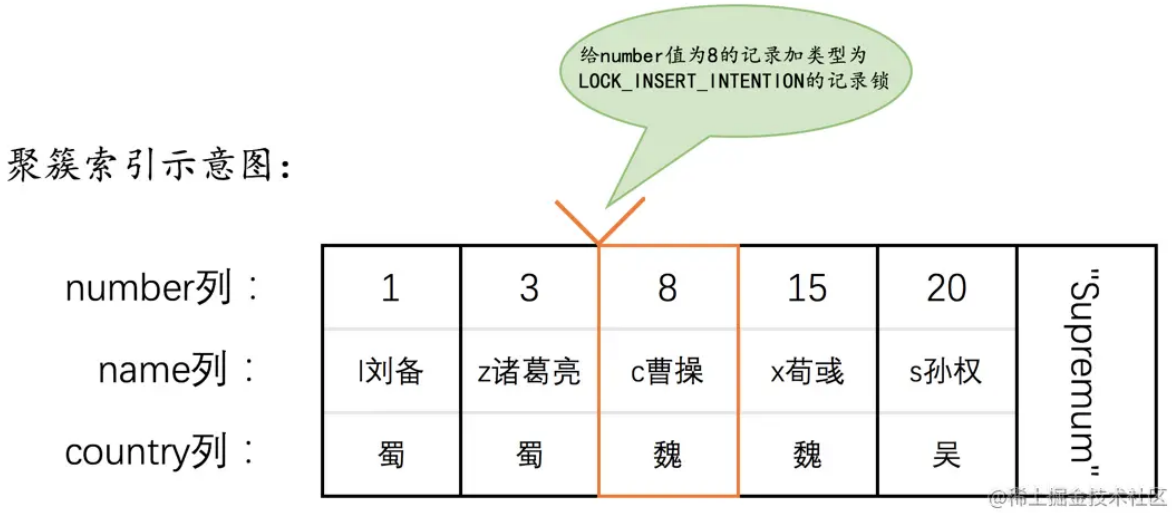


`next-key锁` 的本质就是一个 `正经记录锁` 和一个 `gap锁` 的合体，它既能保护该条记录，又能阻止别的事务将新记录插入被保护记录前边的 `间隙` 。

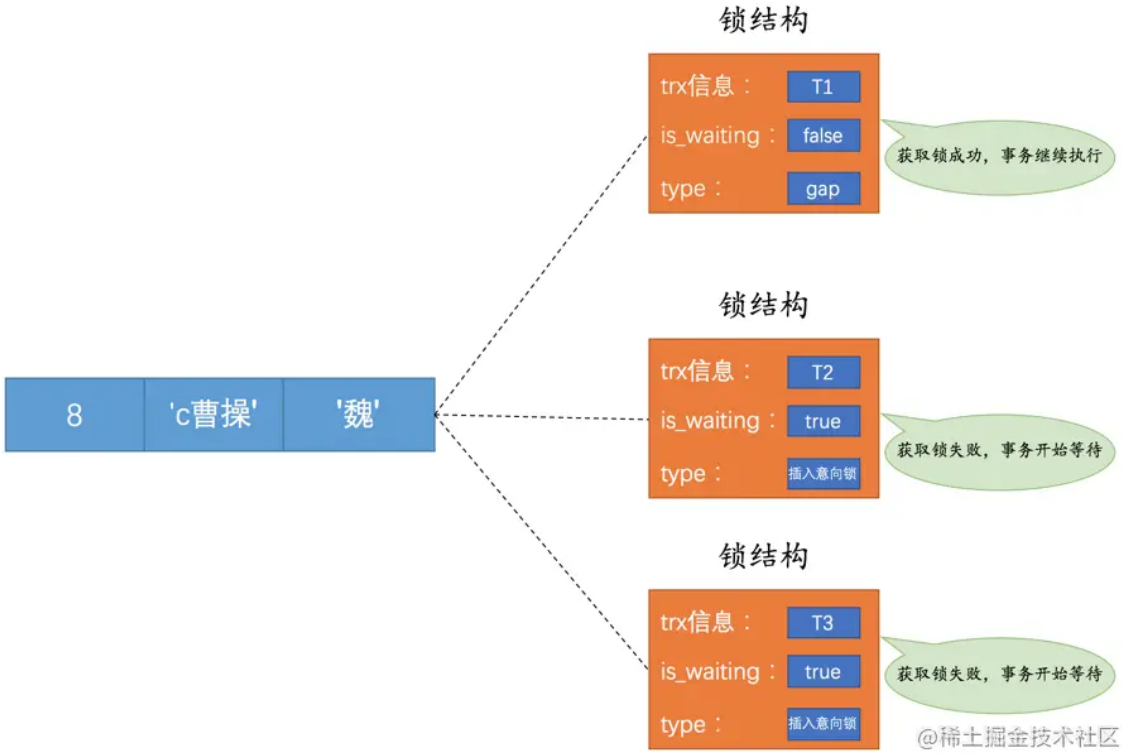
- `Insert Intention Locks` ：

  我们说一个事务在插入一条记录时需要判断一下插入位置是不是被别的事务加了所谓的 `gap锁` （ `next-key锁` 也包含 `gap锁` ，后边就不强调了），如果有的话，插入操作需要等待，直到拥有 `gap锁` 的那个事务提交。但是设计 `InnoDB` 的大叔规定事务在等待的时候也需要在内存中生成一个 `锁结构` ，表明有事务想在某个 `间隙` 中插入新记录，但是现在在等待。设计 `InnoDB` 的大叔就把这种类型的锁命名为 `Insert Intention Locks` ，官方的类型名称为： `LOCK_INSERT_INTENTION` ，我们也可以称为 `插入意向锁` 。

  比方说我们把 `number` 值为 `8` 的那条记录加一个 `插入意向锁` 的示意图如下：



  为了让大家彻底理解这个 `插入意向锁` 的功能，我们还是举个例子然后画个图表示一下。比方说现在 `T1` 为 `number` 值为 `8` 的记录加了一个 `gap锁` ，然后 `T2` 和 `T3` 分别想向 `hero` 表中插入 `number` 值分别为 `4` 、 `5` 的两条记录，所以现在为 `number` 值为 `8` 的记录加的锁的示意图就如下所示：



> 小贴士： 我们在锁结构中又新添了一个type属性，表明该锁的类型。稍后会全面介绍InnoDB存储引擎中的一个锁结构到底长什么样。

从图中可以看到，由于 `T1` 持有 `gap锁` ，所以 `T2` 和 `T3` 需要生成一个 `插入意向锁` 的 `锁结构` 并且处于等待状态。当 `T1` 提交后会把它获取到的锁都释放掉，这样 `T2` 和 `T3` 就能获取到对应的 `插入意向锁` 了（本质上就是把插入意向锁对应锁结构的 `is_waiting` 属性改为 `false` ） ， `T2` 和 `T3` 之间也并不会相互阻塞，它们可以同时获取到 `number` 值为8的 `插入意向锁` ，然后执行插入操作。事实上插入意向锁并不会阻止别的事务继续获取该记录上任何类型的锁（ `插入意向锁` 就是这么鸡肋）。

- 隐式锁

  我们前边说一个事务在执行 `INSERT` 操作时，如果即将插入的 `间隙` 已经被其他事务加了 `gap锁` ，那么本次 `INSERT` 操作会阻塞，并且当前事务会在该间隙上加一个 `插入意向锁` ，否则一般情况下 `INSERT` 操作是不加锁的。那如果一个事务首先插入了一条记录（此时并没有与该记录关联的锁结构），然后另一个事务：

  - 立即使用 `SELECT ... LOCK IN SHARE MODE` 语句读取这条记录，也就是在要获取这条记录的 `S锁` ，或者使用 `SELECT ... FOR UPDATE` 语句读取这条记录，也就是要获取这条记录的 `X锁` ，该咋办？

    如果允许这种情况的发生，那么可能产生 `脏读` 问题。

  - 立即修改这条记录，也就是要获取这条记录的 `X锁` ，该咋办？

    如果允许这种情况的发生，那么可能产生 `脏写` 问题。

  这时候我们前边唠叨了很多遍的 `事务id` 又要起作用了。我们把聚簇索引和二级索引中的记录分开看一下：

  - 情景一：对于聚簇索引记录来说，有一个 `trx_id` 隐藏列，该隐藏列记录着最后改动该记录的 `事务id` 。那么如果在当前事务中新插入一条聚簇索引记录后，该记录的 `trx_id` 隐藏列代表的的就是当前事务的 `事务id` ，如果其他事务此时想对该记录添加 `S锁` 或者 `X锁` 时，首先会看一下该记录的 `trx_id` 隐藏列代表的事务是否是当前的活跃事务，如果是的话，那么就帮助当前事务创建一个 `X锁` （也就是为当前事务创建一个锁结构， `is_waiting` 属性是 `false` ），然后自己进入等待状态（也就是为自己也创建一个锁结构， `is_waiting` 属性是 `true` ）。

  - 情景二：对于二级索引记录来说，本身并没有 `trx_id` 隐藏列，但是在二级索引页面的 `Page Header` 部分有一个 `PAGE_MAX_TRX_ID` 属性，该属性代表对该页面做改动的最大的 `事务id` ，如果 `PAGE_MAX_TRX_ID` 属性值小于当前最小的活跃 `事务id` ，那么说明对该页面做修改的事务都已经提交了，否则就需要在页面中定位到对应的二级索引记录，然后回表找到它对应的聚簇索引记录，然后再重复 `情景一` 的做法。

  通过上边的叙述我们知道，一个事务对新插入的记录可以不显式的加锁（生成一个锁结构），但是由于 `事务id` 这个牛逼的东东的存在，相当于加了一个 `隐式锁` 。别的事务在对这条记录加 `S锁` 或者 `X锁` 时，由于 `隐式锁` 的存在，会先帮助当前事务生成一个锁结构，然后自己再生成一个锁结构后进入等待状态。

  > 小贴士： 除了插入意向锁，在一些特殊情况下INSERT还会获取一些锁，我们稍后唠叨哈。

InnoDB锁的内存结构

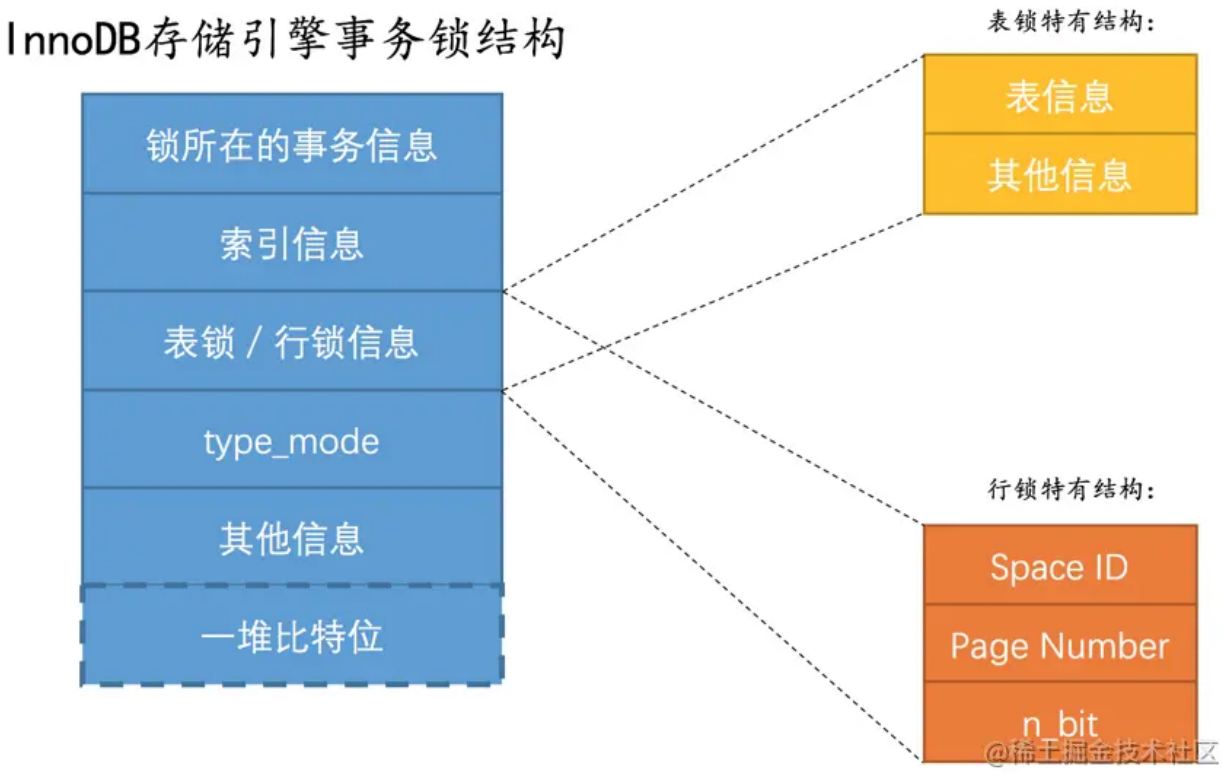
我们前边说对一条记录加锁的本质就是在内存中创建一个 `锁结构` 与之关联，那么是不是一个事务对多条记录加锁，就要创建多个 `锁结构` 呢？比方说事务 `T1` 要执行下边这个语句：

```
# 事务T1
SELECT * FROM hero LOCK IN SHARE MODE;
```

很显然这条语句需要为 `hero表` 中的所有记录进行加锁，那是不是需要为每条记录都生成一个 `锁结构` 呢？其实理论上创建多个 `锁结构` 没问题，反而更容易理解，但是谁知道你在一个事务里想对多少记录加锁呢，如果一个事务要获取10000条记录的锁，要生成10000个这样的结构也太亏了吧！所以设计 `InnoDB` 的大叔本着勤俭节约的传统美德，决定在对不同记录加锁时，如果符合下边这些条件：

- 在同一个事务中进行加锁操作
- 被加锁的记录在同一个页面中
- 加锁的类型是一样的
- 等待状态是一样的

那么这些记录的锁就可以被放到一个 `锁结构` 中。当然，这么空口白牙的说有点儿抽象，我们还是画个图来看看 `InnoDB` 存储引擎中的 `锁结构` 具体长啥样吧：



我们看看这个结构里边的各种信息都是干嘛的：

- `锁所在的事务信息`：

  不论是`表锁`还是`行锁`，都是在事务执行过程中生成的，哪个事务生成了这个`锁结构`，这里就记载着这个事务的信息。

  > 小贴士： 实际上这个所谓的`锁所在的事务信息`在内存结构中只是一个指针而已，所以不会占用多大内存空间，通过指针可以找到内存中关于该事务的更多信息，比方说事务id是什么。下边介绍的所谓的`索引信息`其实也是一个指针。

- `索引信息`：

  对于`行锁`来说，需要记录一下加锁的记录是属于哪个索引的。

- `表锁／行锁信息`：

  `表锁结构`和`行锁结构`在这个位置的内容是不同的：

  - 表锁：

    记载着这是对哪个表加的锁，还有其他的一些信息。

  - 行锁：

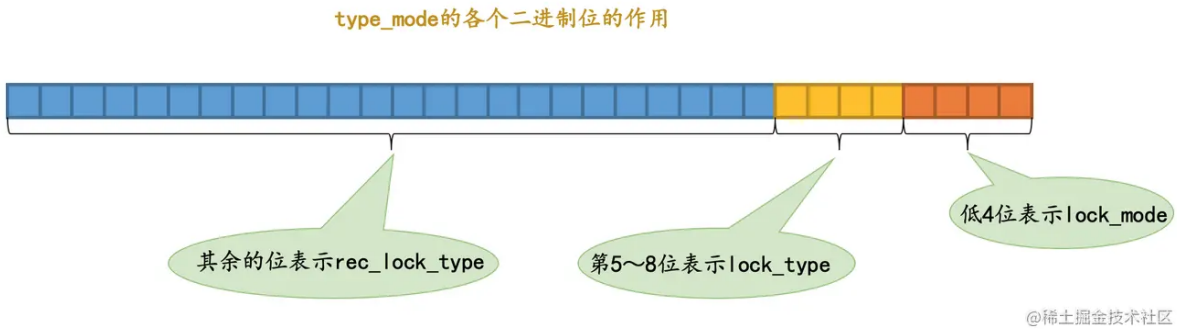
    记载了三个重要的信息：

    - `Space ID`：记录所在表空间。
    - `Page Number`：记录所在页号。
    - `n_bits`：对于行锁来说，一条记录就对应着一个比特位，一个页面中包含很多记录，用不同的比特位来区分到底是哪一条记录加了锁。为此在行锁结构的末尾放置了一堆比特位，这个`n_bits`属性代表使用了多少比特位。

    > 小贴士： 并不是该页面中有多少记录，n_bits属性的值就是多少。为了让之后在页面中插入了新记录后也不至于重新分配锁结构，所以n_bits的值一般都比页面中记录条数多一些。

- `type_mode` ：

  这是一个32位的数，被分成了`lock_mode`、`lock_type`和`rec_lock_type`三个部分，如图所示：



  - 锁的模式（`lock_mode`），占用低4位，可选的值如下：
    - `LOCK_IS`（十进制的`0`）：表示共享意向锁，也就是`IS锁`。
    - `LOCK_IX`（十进制的`1`）：表示独占意向锁，也就是`IX锁`。
    - `LOCK_S`（十进制的`2`）：表示共享锁，也就是`S锁`。
    - `LOCK_X`（十进制的`3`）：表示独占锁，也就是`X锁`。
    - `LOCK_AUTO_INC`（十进制的`4`）：表示`AUTO-INC锁`。

    > 小贴士： 在InnoDB存储引擎中，LOCK_IS，LOCK_IX，LOCK_AUTO_INC都算是表级锁的模式，LOCK_S和LOCK_X既可以算是表级锁的模式，也可以是行级锁的模式。

  - 锁的类型（`lock_type`），占用第5～8位，不过现阶段只有第5位和第6位被使用：
    - `LOCK_TABLE`（十进制的`16`），也就是当第5个比特位置为1时，表示表级锁。
    - `LOCK_REC`（十进制的`32`），也就是当第6个比特位置为1时，表示行级锁。

- 行锁的具体类型（`rec_lock_type`），使用其余的位来表示。只有在`lock_type`的值为`LOCK_REC`时，也就是只有在该锁为行级锁时，才会被细分为更多的类型：
    - `LOCK_ORDINARY`（十进制的`0`）：表示`next-key锁`。
    - `LOCK_GAP`（十进制的`512`）：也就是当第10个比特位置为1时，表示`gap锁`。
    - `LOCK_REC_NOT_GAP`（十进制的`1024`）：也就是当第11个比特位置为1时，表示`正经记录锁`。
    - `LOCK_INSERT_INTENTION`（十进制的`2048`）：也就是当第12个比特位置为1时，表示插入意向锁。
    - 其他的类型：还有一些不常用的类型我们就不多说了。

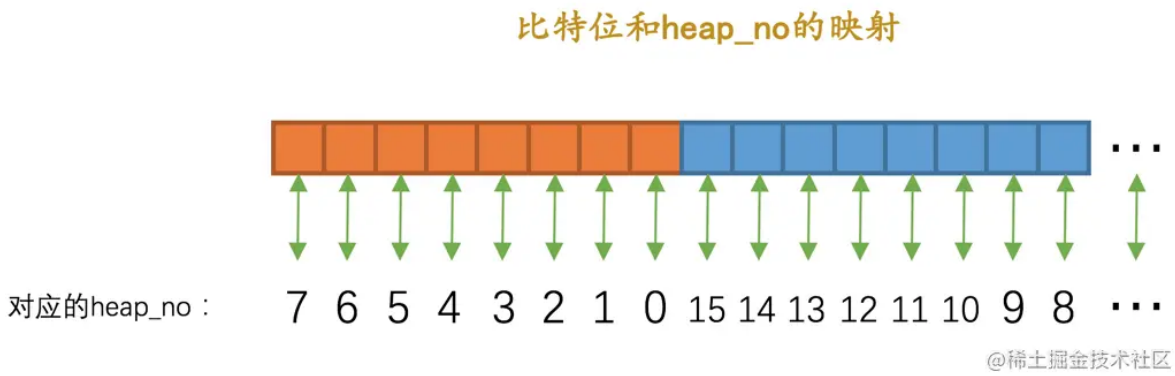
    怎么还没看见`is_waiting`属性呢？这主要还是设计`InnoDB`的大叔太抠门了，一个比特位也不想浪费，所以他们把`is_waiting`属性也放到了`type_mode`这个32位的数字中：

    `LOCK_WAIT`（十进制的`256`）：也就是当第9个比特位置为`1`时，表示`is_waiting`为`true`，也就是当前事务尚未获取到锁，处在等待状态；当这个比特位为`0`时，表示`is_waiting`为`false`，也就是当前事务获取锁成功。
- `其他信息`：

  为了更好的管理系统运行过程中生成的各种锁结构而设计了各种哈希表和链表，为了简化讨论，我们忽略这部分信息哈～

- 一堆比特位：

  如果是行锁结构的话，在该结构末尾还放置了一堆比特位，比特位的数量是由上边提到的 n_bits 属性表示的。我们前边唠叨InnoDB记录结构的时候说过，页面中的每条记录在 记录头信息 中都包含一个 heap_no 属性，伪记录 Infimum 的 heap_no 值为 0 ， Supremum 的 heap_no 值为 1 ，之后每插入一条记录， heap_no 值就增1。 锁结构 最后的一堆比特位就对应着一个页面中的记录，一个比特位映射一个 heap_no ，不过为了编码方便，映射方式有点怪：



  > 小贴士： 这么怪的映射方式纯粹是为了敲代码方便，大家不要大惊小怪，只需要知道一个比特位映射到页内的一条记录就好了。

可能上边的描述大家觉得还是有些抽象，我们还是举个例子说明一下。比方说现在有两个事务 T1 和 T2 想对 hero 表中的记录进行加锁， hero 表中记录比较少，假设这些记录都存储在所在的表空间号为 67 ，页号为 3 的页面上，那么如果：

- `T1` 想对 `number` 值为 `15` 的这条记录加 `S型正常记录锁` ，在对记录加行锁之前，需要先加表级别的 `IS` 锁，也就是会生成一个表级锁的内存结构，不过我们这里不关心表级锁，所以就忽略掉了哈～ 接下来分析一下生成行锁结构的过程：
  - 事务 `T1` 要进行加锁，所以锁结构的 `锁所在事务信息` 指的就是 `T1` 。
  - 直接对聚簇索引进行加锁，所以索引信息指的其实就是 `PRIMARY` 索引。
  - 由于是行锁，所以接下来需要记录的是三个重要信息：
    - `Space ID` ：表空间号为 `67` 。
    - `Page Number` ：页号为 `3` 。
    - `n_bits` ：我们的 `hero` 表中现在只插入了5条用户记录，但是在初始分配比特位时会多分配一些，这主要是为了在之后新增记录时不用频繁分配比特位。其实计算 `n_bits` 有一个公式：

      ```
      n_bits = (1 + ((n_recs + LOCK_PAGE_BITMAP_MARGIN) / 8)) * 8
      ```

      其中 `n_recs` 指的是当前页面中一共有多少条记录（算上伪记录和在垃圾链表中的记录），比方说现在 `hero` 表一共有7条记录（5条用户记录和2条伪记录），所以 `n_recs` 的值就是 `7` ， `LOCK_PAGE_BITMAP_MARGIN` 是一个固定的值 `64` ，所以本次加锁的 `n_bits` 值就是：

      ```
      n_bits = (1 + ((7 + 64) / 8)) * 8 = 72
      ```

    - `type_mode` 是由三部分组成的：
      - `lock_mode` ，这是对记录加 `S锁` ，它的值为 `LOCK_S` 。
      - `lock_type` ，这是对记录进行加锁，也就是行锁，所以它的值为 `LOCK_REC` 。
      - `rec_lock_type` ，这是对记录加 `正经记录锁` ，也就是类型为 `LOCK_REC_NOT_GAP` 的锁。另外，由于当前没有其他事务对该记录加锁，所以应当获取到锁，也就是 `LOCK_WAIT` 代表的二进制位应该是0。

    综上所属，此次加锁的 `type_mode` 的值应该是：

    ```
    type_mode = LOCK_S | LOCK_REC | LOCK_REC_NOT_GAP
    也就是：
    type_mode = 2 | 32 | 1024 = 1058
    ```

  - 其他信息

    略～

- 一堆比特位

  因为 `number` 值为 `15` 的记录 `heap_no` 值为 `5` ，根据上边列举的比特位和 `heap_no` 的映射图来看，应该是第一个字节从低位往高位数第6个比特位被置为1，就像这样：

综上所述，事务 `T1` 为 `number` 值为5的记录加锁生成的锁结构就如下图所示：

- `T2` 想对 `number` 值为 `3` 、 `8` 、 `15` 的这三条记录加 `X型的next-key锁` ，在对记录加行锁之前，需要先加表级别的 `IX` 锁，也就是会生成一个表级锁的内存结构，不过我们这里不关心表级锁，所以就忽略掉了哈～

  现在 `T2` 要为3条记录加锁， `number` 为 `3` 、 `8` 的两条记录由于没有其他事务加锁，所以可以成功获取这条记录的 `X型next-key锁` ，也就是生成的锁结构的 `is_waiting` 属性为 `false` ；但是 `number` 为 `15` 的记录已经被 `T1` 加了 `S型正经记录锁` ， `T2` 是不能获取到该记录的 `X型next-key锁` 的，也就是生成的锁结构的 `is_waiting` 属性为 `true` 。因为等待状态不相同，所以这时候会生成两个 `锁结构` 。这两个锁结构中相同的属性如下：

  - 事务 `T2` 要进行加锁，所以锁结构的 `锁所在事务信息` 指的就是 `T2` 。
  - 直接对聚簇索引进行加锁，所以索引信息指的其实就是 `PRIMARY` 索引。
  - 由于是行锁，所以接下来需要记录是三个重要信息：
    - `Space ID` ：表空间号为 `67` 。
    - `Page Number` ：页号为 `3` 。
    - `n_bits` ：此属性生成策略同 `T1` 中一样，该属性的值为 `72` 。
    - `type_mode` 是由三部分组成的：
      - `lock_mode` ，这是对记录加 `X锁` ，它的值为 `LOCK_X` 。
      - `lock_type` ，这是对记录进行加锁，也就是行锁，所以它的值为 `LOCK_REC` 。
      - `rec_lock_type` ，这是对记录加 `next-key锁` ，也就是类型为 `LOCK_ORDINARY` 的锁。
  - 其他信息

    略～

  不同的属性如下：

- 为 `number` 为 `3` 、 `8` 的记录生成的 `锁结构` ：
  - `type_mode` 值。

    由于可以获取到锁，所以 `is_waiting` 属性为 `false` ，也就是 `LOCK_WAIT` 代表的二进制位被置0。所以：

    ```
    type_mode = LOCK_X | LOCK_REC |LOCK_ORDINARY
    也就是
    type_mode = 3 | 32 | 0 = 35
    ```

  - `一堆比特位`

    因为 `number` 值为 `3` 、 `8` 的记录 `heap_no` 值分别为 `3` 、 `4` ，根据上边列举的比特位和 `heap_no` 的映射图来看，应该是第一个字节从低位往高位数第4、5个比特位被置为1，就像这样：

  综上所述，事务 `T2` 为 `number` 值为 `3` 、 `8` 两条记录加锁生成的锁结构就如下图所示：


T2
PRIMARY
Space ID：67
Page Number：3
n_bits：72
35
其他信息
000110000000…
@稀土掘金技术社区

- 为 `number` 为 `15` 的记录生成的 `锁结构` ：
  - `type_mode` 值。

    由于不可以获取到锁，所以 `is_waiting` 属性为 `true` ，也就是 `LOCK_WAIT` 代表的二进制位被置1。所以：

    ```
    type_mode = LOCK_X | LOCK_REC |LOCK_ORDINARY | LOCK_WAIT
    也就是
    type_mode = 3 | 32 | 0 | 256 = 291
    ```

  - `一堆比特位`

    因为 `number` 值为 `15` 的记录 `heap_no` 值为 `5` ，根据上边列举的比特位和 `heap_no` 的映射图来看，应该是第一个字节从低位往高位数第6个比特位被置为1，就像这样：

综上所述，事务 `T2` 为 `number` 值为 `15` 的记录加锁生成的锁结构就如下图所示：

综上所述，事务 `T1` 先获取 `number` 值为 `15` 的 `S型正经记录锁` ，然后事务 `T2` 获取 `number` 值为 `3` 、 `8` 、 `15` 的 `X型正经记录锁` 共需要生成3个锁结构。噗～ 关于锁结构我本来就想写一点点的，没想到一写起来就停不下了，大家乐呵乐呵看哈～

> 小贴士： 上边事务T2在对number值分别为3、8、15这三条记录加锁的情景中，是按照先对number值为3的记录加锁、再对number值为8的记录加锁，最后对number值为15的记录加锁的顺序进行的，如果我们一开始就对number值为15的记录加锁，那么该事务在为number值为15的记录生成一个锁结构后，直接就进入等待状态，就不为number值为3、8的两条记录生成锁结构了。在事务T1提交后会把在number值为15的记录上获取的锁释放掉，然后事务T2就可以获取该记录上的锁，这时再对number值为3、8的两条记录加锁时，就可以复用之前为number值为15的记录加锁时生成的锁结构了。

语句加锁分析

当然，本文只是解释了 `InnoDB` 存储引擎中各种锁是个啥以及它们对应的内存锁结构，不过相信各位小伙伴对“锁”的概念其实还不是特别明晰，就是具体的某条语句到底应该加哪些锁。哈哈，语句加锁分析是个非常大的命题，如果有同学想深入学习关于“锁”的这一部分知识，请关注我的微信公众号 「我们都是小青蛙」

之后输入关键字“锁”，即可获取三篇语句加锁分析文章，看完有收获一定要来转发吼，么么哒～

> 小贴士： 当然，除了语句加锁分析的三篇文章，还会陆续在公众号中发布相关补充文章，像「我们都是小青蛙」这种清流公众号已经不多了哈～