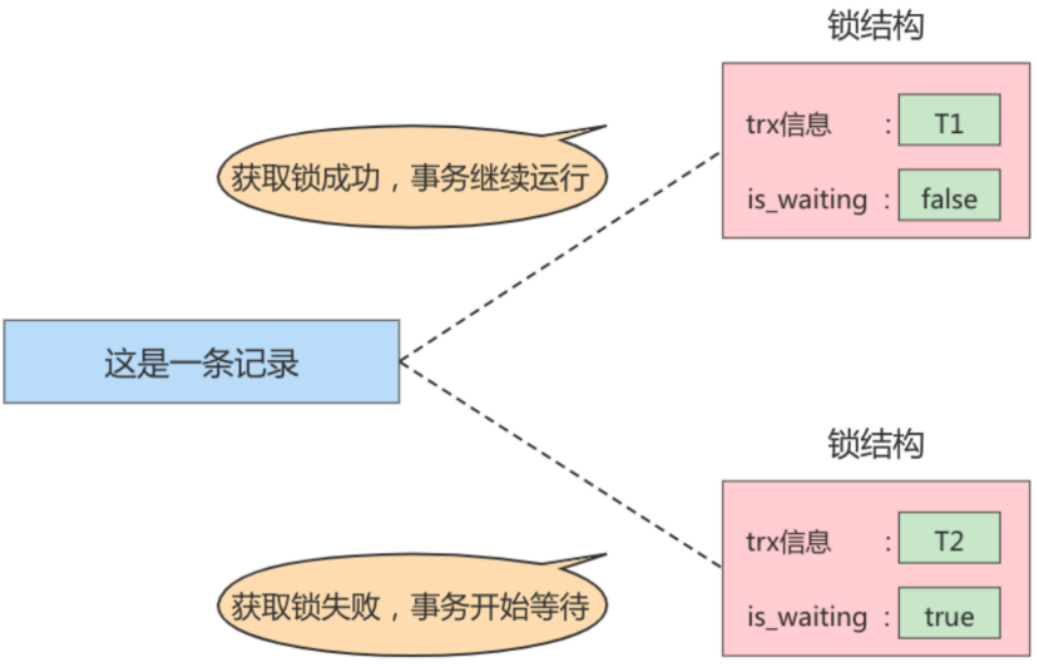
Part 15 锁

1. MySQL并发事务访问相同记录

- 读读
  - 不加锁
- 写写
  - 锁结构

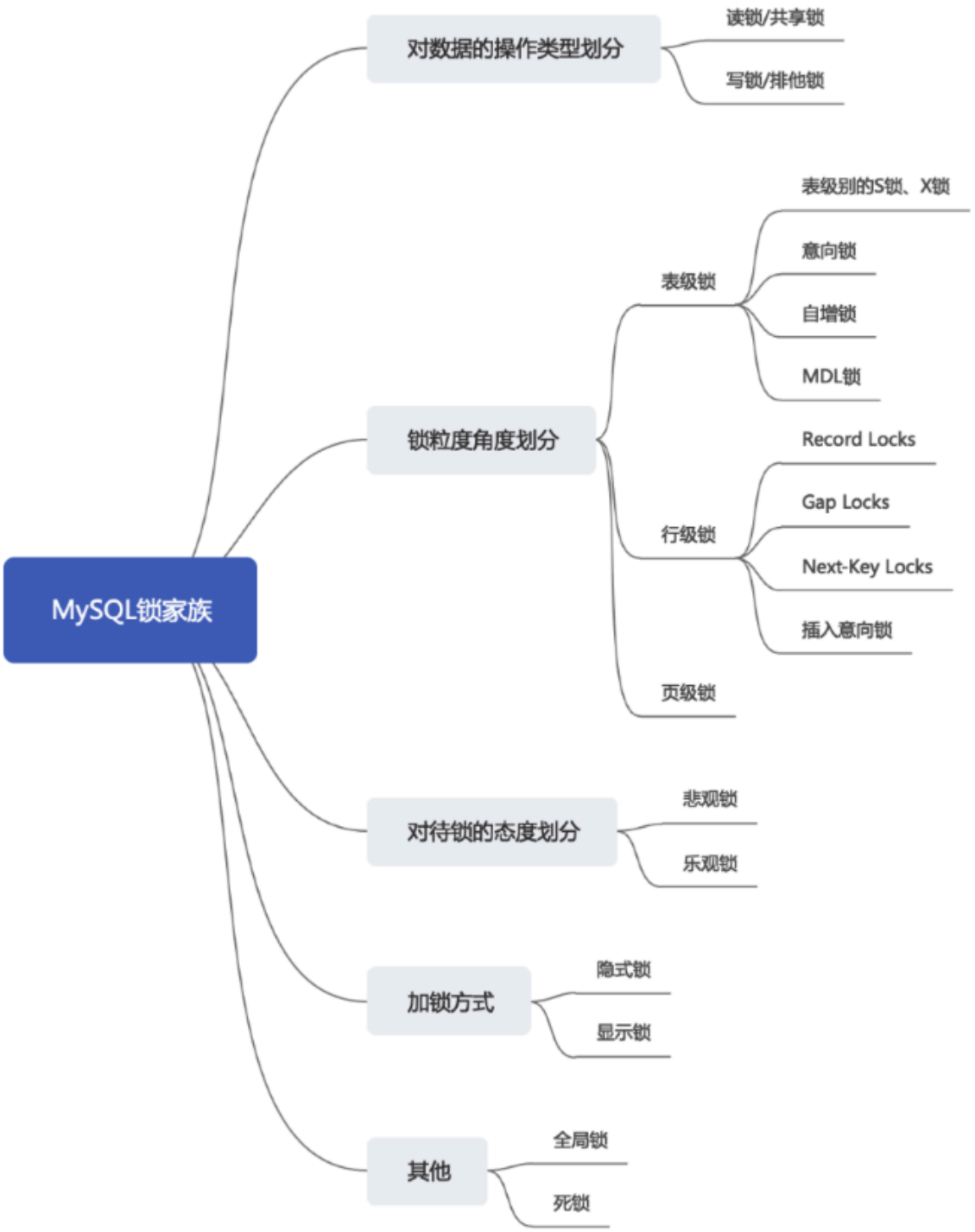


  - 要写的记录没被加锁，获取锁成功
  - 要写的记录已被加锁，获取锁失败
- 读写或写读
  - 可能发生`脏读`、`不可重复读`、`幻读`的问题

解决并发问题

- 方案一：读操作利用多版本并发控制（MVCC，下章讲解），写操作进行加锁。
  - 读就是生成一个ReadView，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。
  - 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。
- 方案二：读、写操作都采用`加锁`的方式。
- 两个方案对比
  - 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。

2. 锁的不同角度分类



2.1 从数据操作的类型划分：读锁、写锁

共享锁(Shared Lock，SLock)和**排他锁(Exclusive Lock，XLock),**也叫读锁(readlock)和写锁(write lock)。

需要注意的是对于 InnoDB 引擎来说，读锁和写锁可以加在表上，也可以加在行上。

	X锁	**S锁**
X锁	不兼容	不兼容
S锁	不兼容	**兼容**

2.1.1 锁定读

有时候需要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。因此有两种特殊的SELECT语句格式：

```
# 对读加S锁
SELECT ... LOCK IN SHARE MODE;
# 或
SELECT ... FOR SHARE;#(8.0新增语法)

# 对读加X锁
SELECT ... FOR UPDATE;
```

MySQL8.0新特性：

在5.7及之前的版本，`SELECT...FOR UPDATE`，如果获取不到锁，会一直等待，直到`innodb_lock_wait_timeout`超时。

在8.0版本中，`SELECT...FOR UPDATE`，`SELECT...FOR SHARE`添加`NOWAIT`、`SKIP LOCKED`语法，跳过锁等待，或者跳过锁定。

2.1.2 写操作

- DELETE
  - 可以看做写锁的锁定读。
- UPDATE
  - 情况一：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`，可以看做写锁的锁定读。
  - 情况二：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。在定位的待修改记录位置可以看做是获取写锁的锁定读，新插入的记录由`INSERT`操作提供的`隐式锁`进行保护。
  - 情况三：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，通过一种称之为`隐式锁`的结构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`。

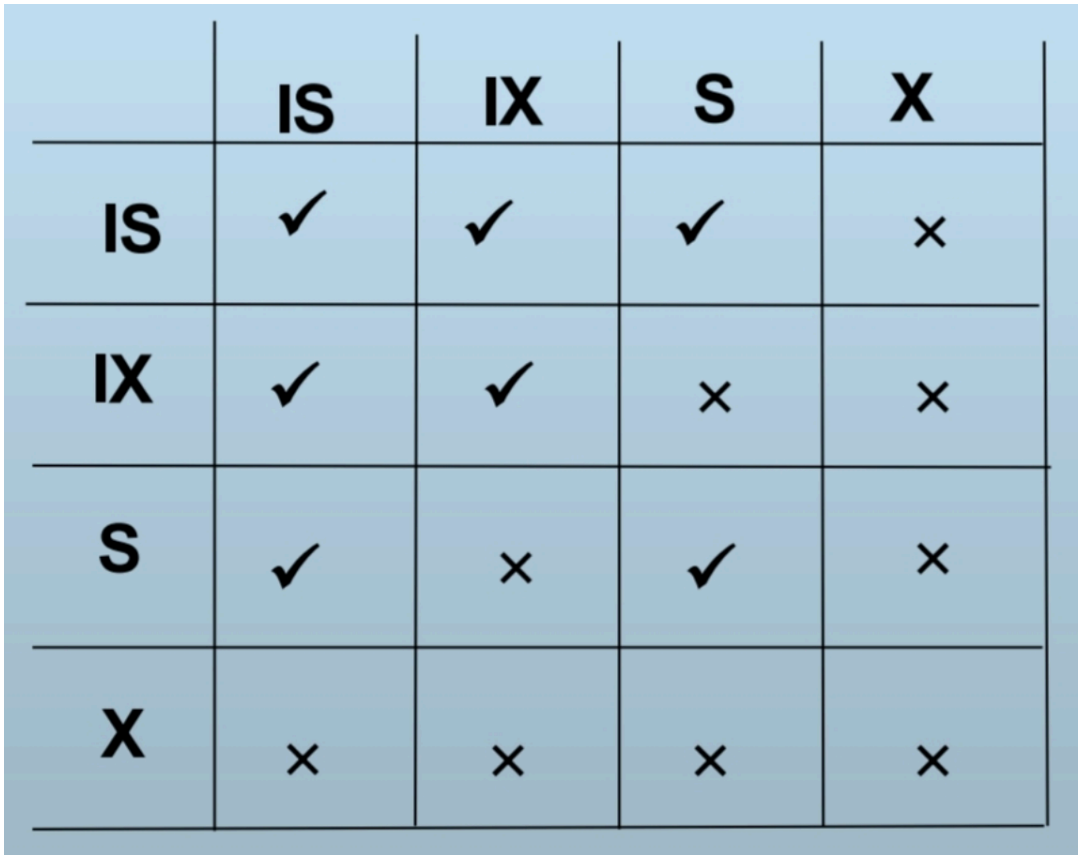
2.2 从数据操作的粒度划分：表级锁、页级锁、行锁

2.2.1 表锁（Table Lock）

锁定整张表，加锁的开销小，可以很好的避免死锁问题，但粒度大导致并发率低。

- 表级别的S锁、X锁
  - MyISAM在执行查询语句（SELECT）前，会给涉及的所有表加读锁，在执行增删改操作前，会给涉及的表加写锁。但InnoDB不会。

- 一般情况下，不会使用InnoDB存储引擎提供的`表级别`的`S锁`和`X锁`。只会在一些特殊情况下，比方说`崩溃恢复`过程中用到。比如，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。
- 要尽量避免手动锁表，并不会提供什么额外的保护，只是会降低并发能力而已。
锁类型	自己可读	自己可写	自己可操作其他表	他人可读	他人可写
读锁	是	否	否	是	否，等
写锁	是	是	否	否，等	否，等

- 意向锁（intention lock）
  - InnoDB 支持`多粒度锁（multiple granularity locking）`，它允许行级锁与表级锁共存，而 **意向锁** 就是其中的一种`表锁`。
  - **意向共享锁 （intention shared lock, IS）** ：事务有意向对表中的某些行加共享锁 （S锁）。如果事务想要获得数据表中某些记录的共享锁，就需要在数据表上`添加意向共享锁`。
  - **意向排他锁 （intention exclusive lock, IX）** ：事务有意向对表中的某些行加排他锁（X锁）。如果事务想要获得数据表中某些记录的排他锁，就需要在数据表上`添加意向排他锁`。
  - 意向锁是由存储引擎`自己维护的`，用户无法手动操作意向锁，在为数据行加共享 / 排他锁之前，InooDB 会先获取该数据行`所在数据表的对应意向锁`。
  - 

  - 意向锁的并发性：意向锁不会与行级的共享 / 排他锁互斥！意向锁在保证并发性的前提下，实现了`行锁和表锁共存`且`满足事务隔离性`的要求。
- 自增锁（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。如果某字段声明了AUTO_INCREMENT，意味着在书写插入语句时不需要为其赋值。
  - 所有插入数据的方式

        - 方式一：“Simple inserts”（简单插入）

          可以`预先确定要插入的行数`（当语句被初始处理时）的语句。包括没有嵌套子查询的单行和多行`INSERT...VALUES()`和`REPLACE`语句。

        - 方式二：“Bulk inserts”（批量插入）

          `事先不知道要插入的行数`（和所需自动递增值的数量）的语句。比如INSERT ... SELECT，REPLACE... SELECT和LOAD DATA语句，但不包括纯INSERT。 InnoDB在每处理一行，为AUTO_INCREMENT列分配一个新值。

        - 方式三：“Mixed-mode inserts”（混合模式插入）

          这些是“Simple inserts”语句但是指定部分新行的自动递增值。例如INSERT INTO teacher (id,name) VALUES (1,'a'), (NULL,'b'), (5,'c'), (NULL,'d');只是指定了部分id的值。另一种类型的“混合模式插入”是`INSERT ... ON DUPLICATE KEY UPDATE`。

    - 对于上面数据插入的案例，MySQL中采用了`自增锁`的方式来实现，**AUTO-INC锁是当向使用含有AUTO_INCREMENT列的表中插入数据时需要获取的一种特殊的表级锁**，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。**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。也正因为此，其并发性显然并不高，**当我们向一个有AUTO_INCREMENT关键字的主键插入值的时候，每条语句都要对这个表锁进行竞争**，这样的并发潜力其实是很低下的，所以innodb通过`innodb_autoinc_lock_mode`的不同取值来提供不同的锁定机制，来显著提高SQL语句的可伸缩性和性能。

    - `innodb_autoinc_lock_mode`

        - 值为0时：传统的锁定模式。
        - 值为1时：批量插入需要AUTO-INC表级锁，简单插入因为已知插入数量，在轻量锁的控制下获得所需数量的自动递增值，它只在分配过程的持续时间内保持，而不是直到语句完成。
        - 值为2时：所有类INSERT语句都不会使用表级AUTO-INC锁，并且可以同时执行多个语句。从 MySQL 8.0 开始，交错锁模式是`默认`设置。这是最快和最可扩展的锁定模式，但是当使用基于语句的复制或恢复方案时，`从二进制日志重播SQL语句时，这是不安全的。(主从复制id可能不一致)`。可以保证所有都是自动递增值，但不保证对同一个事务值是连续的。

- 元数据锁（MDL锁）
    - MySQL5.5引入了meta data lock，简称MDL锁，属于表锁范畴。MDL 的作用是，保证读写的正确性。
    - 比如，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个`表结构做变更`，增加了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。
    - **当对一个表做增删改查操作的时候，加 MDL读锁；当要对表做结构变更操作的时候，加 MDL写锁。**读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性，解决了DML和DDL操作之间的一致性问题。`不需要显式使用`，在访问一个表的时候会被自动加上。

2.2.2 InnoDB中的行锁

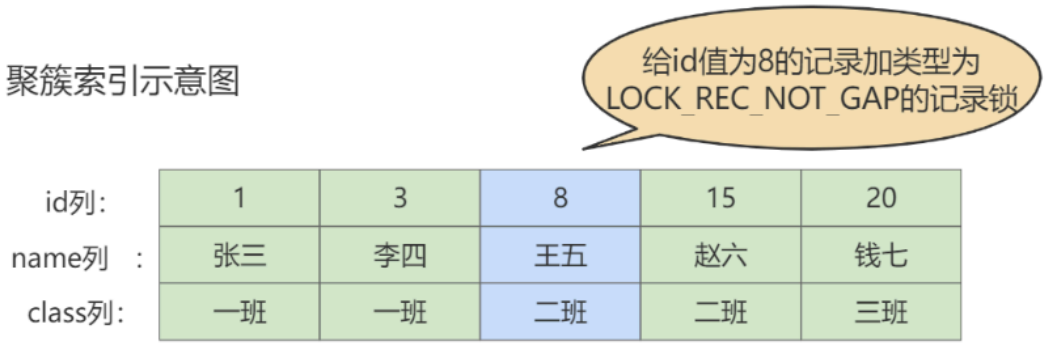
MySQL服务器层并没有实现行锁机制，**行级锁只在存储引擎层实现。**

优点：锁定力度小，`发生锁冲突概率低`，可以实现的`并发度高`。

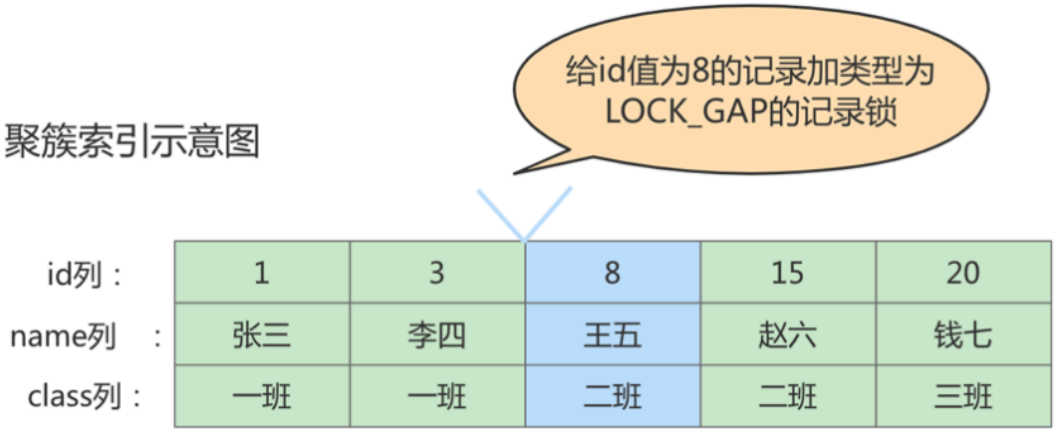
缺点：对于`锁的开销比大`，加锁会比较慢，容易出现`死锁`情况。

InnoDB与MyISAM的最大不同有两点：一是支持事务(TRANSACTION)；二是采用了行级锁。

- 记录锁（Record Locks）
  - 记录锁也就是仅仅把一条记录锁上



  - S型记录锁（和普通读锁规则相同）
  - X型记录锁（与普通写锁规则相同）
- 间隙锁（Gap Locks）
  - **gap锁的提出仅仅是为了防止插入幻影记录而提出的。**



    上图中给id值为8的记录加了gap锁，则不允许在该记录前方的间隙插入新记录。
  - 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是相同的。而且如果对一条记录加了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记录锁或者继续加gap锁。
  - 举例
    - 如果表中没有id = 5的记录

```
select *from student where id =5 lock in share mode;
select * from student where id =5 for update;
```

      以上两句并不冲突，会加间隙锁，不允许在id=5上下两条记录之间插入值。

```
insert into student(id, name, class) values (6, 'tom', '三班');
```

      这一句会冲突。
- 临键锁（Next-Key Locks）
  - 本质就是记录锁和间隙锁的合体。

    - 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InnoDB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。Next-Key Locks 是在存储引擎innodb、事务级别在 可重复读 的情况下使用的数据库锁，innodb默认的锁就是Next-Key locks。
- 插入意向锁（Insert Intention Locks）
    - 不是意向锁
    - **InnoDB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。
    - 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。**

2.2.3 页锁

位置介于行锁和表锁之间，开销也是，并发度一般。

每个层级的锁数量是有限制的，因为锁会占用内存空间，锁空间的大小是有限的。当某个层级的锁数量超过了这个层级的阈值时，就会进行 锁升级。锁升级就是用更大粒度的锁替代多个更小粒度的锁，比如 InnoDB 中行锁升级为表锁，这样做的好处是占用的锁空间降低了，但同时数据的并发度也下降了。

2.3 从对待锁的态度划分:乐观锁、悲观锁

2.3.1 悲观锁（Pessimistic Locking）

对数据被其他事务的修改持保守态度，会通过数据库自身的锁机制来实现，从而保证数据操作的排它性。共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程

Java中 synchronized 和 ReentrantLock 等独占锁就是悲观锁思想的实现。

2.3.2 乐观锁（Optimistic Locking）

乐观锁认为对同一数据的并发操作不会总发生，属于小概率事件，不用每次都对数据上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，也就是 **不采用数据库自身的锁机制，而是通过程序来实现** 。

在程序上，我们可以采用 版本号机制 或者 CAS机制 实现。

乐观锁适用于多读的应用类型，这样可以提高吞吐量 。

在Java中java.util.concurrent.atomic包下的原子变量类就是使用了乐观锁的一种实现方式：CAS实现的。

- 乐观锁的版本号机制
    - 在表中设计一个 版本字段 version，第一次读的时候，会获取 version 字段的取值。然后对数据进行更新或删除操作时，会执行 UPDATE ... SET version = version + 1 WHERE version=version。此时如果已经有事务对这条数据进行了更改，修改就不会成功。
- 乐观锁的时间戳机制
    - 时间戳和版本号机制一样，也是在更新提交的时候，将当前数据的时间戳和更新之前取得的时间戳进行比较，如果两者一致则更新成功，否则就是版本冲突。你能看到乐观锁就是程序员自己控制数据并发操作的权限，基本是通过给数据行增加一个戳（版本号或者时间戳），从而证明当前拿到的数据是否最新。

2.3.3 两种锁的适用场景

- 乐观锁适用于读操作多的，程序实现
- 悲观锁适用于写操作多的，数据库层面实现

2.4 按加锁的方式划分：显式锁、隐式锁

2.4.1 隐式锁

看不懂。。。

当事务需要加锁的时，如果这个锁不可能发生冲突，InnoDB会跳过加锁环节，这种机制称为隐式锁。隐式锁是InnoDB实现的一种延迟加锁机制，其特点是只有在可能发生冲突时才加锁，从而减少了锁的数量，提高了系统整体性能。另外，隐式锁是针对被修改的B+ Tree记录，因此都是记录类型的锁，不可能是间隙锁或Next-Key类型。

隐式锁主要用在插入场景中。在Insert语句执行过程中，必须检查两种情况，一种是如果记录之间加有间隙锁，为了避免幻读，此时是不能插入记录的，另一中情况如果Insert的记录和已有记录存在唯一键冲突，此时也不能插入记录。除此之外，insert语句的锁都是隐式锁，但跟踪代码发现，insert时并没有调用lock_rec_add_to_queue函数进行加锁, 其实所谓隐式锁就是在Insert过程中不加锁。

只有在特殊情况下，才会将隐式锁转换为显示锁。这个转换动作并不是加隐式锁的线程自发去做的，而是其他存在行数据冲突的线程去做的。例如事务1插入记录且未提交，此时事务2尝试对该记录加锁，那么事务2必须先判断记录上保存的事务id是否活跃，如果活跃则帮助事务1建立一个锁对象，而事务2自身进入等待事务1的状态。

判断隐式锁是否存在：

- InnoDB的每条记录中都一个隐含的trx_id字段，这个字段存在于聚集索引的B+Tree中。假设只有主键索引，则在进行插入时，行数据的trx_id被设置为当前事务id；假设存在二级索引，则在对二级索引进行插入时，需要更新所在page的max_trx_id。
- 因此对于主键，只需要通过查看记录隐藏列trx_id是否是活跃事务就可以判断隐式锁是否存在。 对于对于二级索引会相对比较麻烦，先通过二级索引页上的max_trx_id进行过滤，如果无法判断是否活跃则需要通过应用undo日志回溯老版本数据，才能进行准确的判断。

隐式锁的逻辑过程如下：

A. InnoDB的每条记录中都一个隐含的trx_id字段，这个字段存在于聚簇索引的B+Tree中。

B. 在操作一条记录前，首先根据记录中的trx_id检查该事务是否是活动的事务(未提交或回滚)。如果是活动的事务，首先将隐式锁转换为显式锁(就是为该事务添加一个锁)。

C. 检查是否有锁冲突，如果有冲突，创建锁，并设置为waiting状态。如果没有冲突不加锁，跳到E。

D. 等待加锁成功，被唤醒，或者超时。

E. 写数据，并将自己的trx_id写入trx_id字段。

2.4.2 显式锁

通过特定的语句进行加锁，我们一般称之为显示加锁，例如：

显示加共享锁：

```
select ....  lock in share mode
```

显示加排它锁：

```
select ....  for update
```

2.5 其它锁之：全局锁

全局锁就是对`整个数据库实例`加锁。当你需要让整个库处于`只读状态`的时候，可以使用这个命令，之后其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。全局锁的典型使用`场景`是：做`全库逻辑备份`。

全局锁的命令：

```
Flush tables with read lock
```

2.6 死锁

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环。

当出现死锁以后，有`两种策略`：

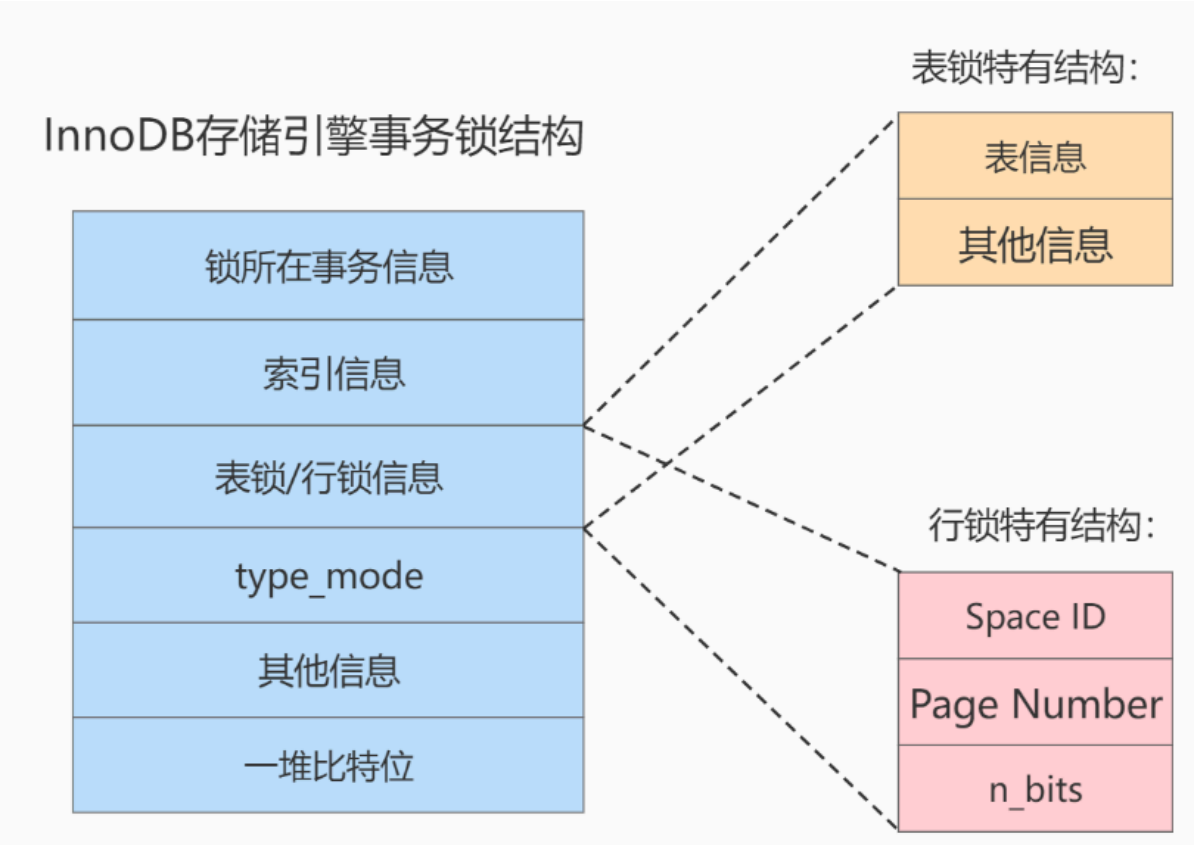
- 一种策略是，直接进入等待，直到超时。这个超时时间可以通过参数`innodb_lock_wait_timeout`来设置。
- 另一种策略是，发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务（将持有最少行级排他锁的事务进行回滚），让其他事务得以继续执行。将参数`innodb_deadlock_detect`设置为on，表示开启这个逻辑。

第二种策略的成本分析

方法 1：如果你能确保这个业务一定不会出现死锁，可以临时把死锁检测关掉。 但是这种操作本身带有一定的风险，因为业务设计的时候一般不会把死锁当做一个严重错误，毕竟出现死锁了，就回滚，然后通过业务重试一般就没问题了，这是`业务无损的`。而关掉死锁检测意味着可能会出现大量的超时，这是`业务有损`的。

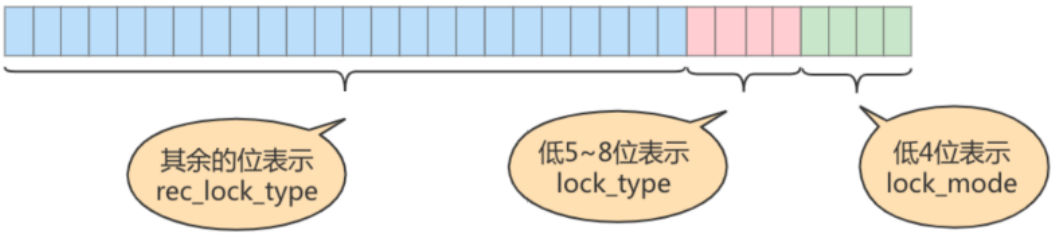
方法 2：控制并发度。 如果并发能够控制住，比如同一行同时最多只有 10 个线程在更新，那么死锁检测的成本很低，就不会出现这个问题。这个并发控制要做在`数据库服务端`。如果你有中间件，可以考虑在`中间件实现`；甚至有能力修改MySQL源码的人，也可以做在MySQL里面。基本思路就是，对于相同行的更新，在进入引擎之前排队，这样在InnoDB内部就不会有大量的死锁检测工作了。

3. 锁的内存结构



- 锁所在的事务信息
  - 此`锁所在的事务信息`在内存结构中只是一个指针，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等。
- 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。这里也是一个指针。
- 表锁 / 行锁信息
  - 表锁：记载着是对哪个表加的锁，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。
      - n_bits的值一般都比页面中记录条数多一些。主要是为了之后在页面中插入了新记录后也不至于重新分配锁结构
- 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

type_mode的各个二进制位的作用



- 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。

  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。
- is_waiting属性：基于内存空间的节省，所以把is_waiting属性放到了type_mode这个 32位的数字中。**LOCK_WAIT**（十进制的 256 ）：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。
- 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。InnoDB数据页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，即一个比特位映射到页内的一条记录。

4. 锁监控

关于MySQL锁的监控，我们一般可以通过检查`InnoDB_row_lock`等状态变量来分析系统上的行锁的争夺情况。

- Innodb_row_lock_current_waits：当前正在等待锁定的数量；
- Innodb_row_lock_time：从系统启动到现在锁定总时间长度；（等待总时长）
- Innodb_row_lock_time_avg：每次等待所花平均时间；（等待平均时长）
- Innodb_row_lock_time_max：从系统启动到现在等待最长的一次所花的时间；
- Innodb_row_lock_waits：系统启动后到现在总共等待的次数；（等待总次数）

其他监控方法：

MySQL把事务和锁的信息记录在了`information_schema`库中，涉及到的三张表分别是`INNODB_TRX`、`INNODB_LOCKS`和`INNODB_LOCK_WAITS`。

`MySQL5.7`及之前，通过information_schema.INNODB_LOCKS，只能看到阻塞事务的锁。

MySQL8.0删除了information_schema.INNODB_LOCKS，添加了`performance_schema.data_locks`，可以通过performance_schema.data_locks查看事务的锁情况，不止可以看到阻塞该事务的锁，还可以看到该事务所持有的锁。

同时，information_schema.INNODB_LOCK_WAITS也被`performance_schema.data_lock_waits`所代替。

5. 附录

没看，找原笔记吧，太难了。