**InnoDB的“基本锁”**

InnoDB是一个事务型的数据库引擎，支持ACID事务，基于这个特征，InnoDB是最核心的功能是行锁，当然InnoDB也提供表锁，但这并非其核心的锁，只是在做数据库运维操作时候辅助使用。InnoDB提供行锁主要有两种，一种是排他锁，一种是共享锁。下面我们来分别介绍一下这两种锁。

**排他锁**又称写锁。当一个事务对某几个上写锁时，不允许其他事务写，但允许读。更不允许其他事务给这几行上任何锁，包括写锁。排他锁除了数据库自身事务处理过程中自动加锁外，我们还可以主动的进行加锁，例如查询过程中加锁方式为： SELECT ... FOR UPDATE。

**共享锁**又称读锁。当一个事务对某几行上读锁时，允许其他事务对这几行进行读操作，但不允许其进行写操作，也不允许其他事务给这几行上排他锁，但允许上读锁。实际上，排他锁已经可以解决并发的问题，然后却存在严重的性能问题，很多场景都是要求并发读，然后却有数据一致性的要求，因而提出了共享锁的概念。后面在mvcc一节中我们会介绍到。同样的共享锁使用过程中我们可以为查询结果主动加锁：SELECT .... LOCK In SHARE MODE。

InnoDB通过这种加锁的方式（主要为排他锁和共享锁）对数据库的访问进行并发控制，又称之为LBCC（基于锁的并发控制）。它的最大好处便是，读不加锁，读写不冲突。既兼顾了数据一致性要求，又达到了可接受的并发性。另外，InnoDB也提供了表级排他锁和表级别共享锁，这里不做过多介绍。

**InnoDB的“辅助锁”**

**表锁**是指锁定一张表，虽然InnoDB主要功能在于行锁，但仍然提供了表锁的功能，开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，但并发度低，本身数据库因为持久性要求已经牺牲了很多性能，如果频繁使用表锁，在高并发情况下会严重影响数据库的性能，故虽然InnoDB提供了表锁功能，但一般除了一些数据库运维操作会主动加表锁，其他不推荐使用表锁。另外表锁与事务无关，当我们提交事务后并不能主动释放锁，而是需要手动释放：LOCK TABLE t1 WRITE, t2 READ; UNLOCK TABLES;

**意向锁**主要是为了解决表锁和行锁加锁可能出现冲突而产生的锁，意向锁分为意向共享锁和意向排他锁。当事务打算给某一行加锁时候会首先取得该表的意向共享锁或者意向排他锁，然后才能对表里的某一行进行加锁操作。同样，当我们申请表锁的时候，会先检查该表的意向锁情况，如果该表存在冲突的意向锁，那么申请加锁过程就会阻塞等待。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 是否冲突 | X（排他锁） | IX（意向排他锁） | S（共享锁） | IS（意向排他锁） |
| X（排他锁） | 冲突 | 冲突 | 冲突 | 冲突 |
| IX（意向排他锁） | 冲突 | 兼容 | 冲突 | 兼容 |
| S（共享锁） | 冲突 | 冲突 | 兼容 | 兼容 |
| IS（意向排他锁） | 冲突 | 兼容 | 兼容 | 兼容 |

**InnoDB的“衍生锁”**

**间隙锁**是一个条件范围锁。InnoDB提供的行锁（行排他锁和行共享锁）解决了某一行的数据一致性问题，但如果只有这两种锁，当我们进行范围查询或者操作时候是无法确保数据一致性的，换句话说，如果没有间隙锁，InnoDB在并发时候的幻读问题是很难解决的。现在通过间隙锁+快照读的方式，InnoDB在RR隔离级别下部分解决了幻读问题。

**临键锁**是一种特殊的间隙锁，在我们使用间隙锁的时候，InnoDB规定了一个左开右闭的区间。例如，我们通过ID索引查询某个范围的用户，范围为大于n，小于m。那么间隙锁的加锁范围包含m，不包含n，即(n,m]，对m所加的锁我们称之为临键锁。

**悲观锁**并不是InnoDB提供的原生概念，而是在开发过程中衍生出的概念。悲观锁是一个主动加锁过程，在事务操作过程中我们对表中的数据加行锁，只有在提交事务后，锁才会主动释放。

**乐观锁**是相对于悲观锁而言的，乐观锁不会主动加锁，而是利用数据库的DML过程被动加锁，乐观锁在大事务场景时候可以极大减少加锁时间，提高并发性能。但是乐观锁也同样有其弊端，当大量操作相同行的并发请求时，会出现大量请求失败的情况，这需要我们尤其注意。

**再谈事务**

我们不罗列概念，我们思考一下，为什么需要事务，那么首先我们就要问，关系型数据库要解决什么问题。事务是MySql最重要的概念，因为默认所有操作都是通过事务完成的，我们知道mysql默认是开启了自动提交事务的。

关系型数据库往往是一个系统数据存储的核心，无法容忍数据丢失问题，即使牺牲了很多性能问题，这就意味着，一个事务提交完成，那么这些数据必须是数据持久化到磁盘上的，来保证数据不丢失，这就是事务的持久性；另外，在数据读写过程中要求不能出现“脏数据”的情况，一个完整的事务要保证整个事务的完整性，一个事务提交完成，只有失败和成功两种结果，不能有中间状态，大多数业务系统是不能接受数据只改变了一部分这种情况的，这就要求事务必须保证其原子性，失败的时候可以自动“退回”到操作前的状态；但操作成功之后我们看到的改变必须是一致的、正确的结果，没有业务系统会愿意接受一个错误的结果，这就要求在事务操作过程中，每个操作都必须是有效的，符合预期目标的，这就是事务的一致性要求；另外，数据库都是支持并发的，事务要求在并发事务之间相互影响尽量小，最好是没有影响，这就是事务的隔离性要求。

为了解决这些问题，InnoDB除了提供两种基本的锁，还衍生出一系列的锁，除此之外，还将事务等级进行了分类，为什么要进行事务的分类呢，因为往往一致性和并发性能是冲突的，越高的一致性要求往往意味着越低的并发性能，为了两者兼顾，数据库提出了事务等级分类，不同需求场景的用户选择不同的事务等级，最大程度的提升数据库的性能。数据库很多复杂的设计都是为了性能与一致性或者持久性兼顾，比如前面提到的意向锁、乐观锁等，还比如后面我们将要介绍的MVCC以及redo log。

**什么是MVCC**

MVCC又称多版本并发控制，MVCC只适用于Msyql隔离级别中的读已提交（Read Committed）和可重复读（Repeatable Read）。对于Read Uncommitted并没有使用到MVCC，其通过直接操作表来完成事务过程，并不会通过快照方式。而串行化事务操作是通过表锁来进行的，表锁会屏蔽掉所有的查询和修改操作，故不存在并发问题，也未使用到MVCC。

RC和RR的事务原理基本一致，唯一的不同是RR增加了间隙锁，解决了幻读问题。MVCC的实现实际上是通过每行提供的两个隐藏字段和undo log来实现的。



每当我们增加或者修改某一行或者对该行加锁的时候，InnoDB事务管理模块会将事务ID添加到该行的隐藏字段中，同时生成一个指向undo log的指针信息也添加到隐藏字段中。如果事务执行不成功，则通过这些记录信息和undo log进行事务的回滚。

**redo log和undo log**

redo log主要是为了解决数据库性能问题而产生的，可以设想一下，如果我们对一行表的索引列进行频繁的增删改操作，每次操作都要进行一次索引查询甚至重建操作，一来会导致数据库性能严重下降，二来有些对索引改动较大的操作将会等待较长时间。严重影响使用体验，因此InnoDB提供了redo log，每次事务提交后会写入到redo log中，在数据库集中或者空闲时间再进行数据库表的实际操作。

而undo log一方面是为了解决性能问题，另一方面是为了方面事务失败后的回滚操作，如MVCC一节中我们提到的，每一行数据在进行增删改操作时候会在undo log中生成一个相反的操作，然后隐藏列中回滚指针会指向该undo log相应的地址，同时将事务操作过程存储到redo log中。这样，如果事务出现异常，InnoDB事务管理模块调用undo log信息将redo log信息清理掉即可。

无论是回滚还是执行事务操作，InnoDB都不会直接对表进行操作，而是提交给redo log和undo log。而最终，他们的执行都会通过purge线程处理。purge线程主要有以下几个作用：

* 清理undo段信息，执行真正删除物理删除。在执行delete或update操作时，实际旧记录没有真正删除，只是在记录上打了一个标记，而是在事务提交后，purge线程真正删除，释放物理页空间。因此，提交过程中会将undo信息加入purge列表，供purge线程处理。
* 释放锁资源，MySql通过锁互斥机制保证不同事务不同时操作一条记录，事务执行后才会真正释放所有锁资源，并唤醒等待其锁资源的其他事务。
* 刷redo日志， MySql实现事务一致性和持久性的机制。通过redo日志落盘操作，保证了即使修改的数据页没有即使更新到磁盘，只要日志是完成了，就能保证数据库的完整性和一致性；
* 清理保存点列表，每个事务实际都会有一个保存点，保存点作用是为了可以回滚到事务的任何一个语句执行前的状态，由于事务都已经提交了，所以保存点列表可以被清理了。

**更好的使用事务**

基于以上特点我们对事务的使用做了一些总结：

* 建立索引，常用查询条件列一定要建立索引，InnoDB提供的事务虽然是基于行锁，但是我们要注意，如果没有索引，我们的查询就过滤就变成了全表扫描，这时候InnoDB加的是表锁。同样，like、is null、<>、字符串忘记加单引号等等也是类似情况，这种要尽量避免。
* 避免大事务操作，一个事务操作过程中我们经常需要主动或者被动的加锁，大的事务操作会导致锁的长时间占用，尤其是范围查询，长时间的锁占用会严重降低数据库的性能，这是在事务操作过程中要尽量避免的。我们应当将大的事务尽量拆解或者分批提交。
* 锁的颗粒度尽量小，尽量少使用表锁或者间隙锁（即尽量少范围查询），这样导致的问题一个是锁争用问题，另一个是多行加锁导致死锁问题，我们应尽量只对我们要操作的数据加锁，而不是随意扩大锁的范围。