MySQL是互联网公司用的最多的数据库，InnoDB是MySQL用的最多的存储引擎，它非常适合大数据量，高并发量的互联网业务。

为何InnoDB能够支撑如此之高的并发，它的内核设计逻辑究竟是什么，今天和大家聊聊InnoDB的**并发控制**，**锁**，**MVCC**。

来龙去脉，容我娓娓道来。

画外音：文章有点长，欢迎提前收藏。

第一节、并发控制

为啥要进行并发控制？

并发的任务对同一个临界资源进行操作，如果不采取措施，可能导致不一致，故必须进行**并发控制**（Concurrency Control）。

技术上，通常如何进行并发控制？

通过并发控制保证数据一致性的常见手段有：

（1）锁（Locking）；

（2）数据多版本（Multi Versioning）；

第二节、锁

如何使用普通锁保证一致性？

普通锁，被使用最多：

（1）操作数据前，锁住，实施互斥，不允许其他的并发任务操作；

（2）操作完成后，释放锁，让其他任务执行；

如此这般，来保证一致性。

普通锁存在什么问题？

简单的锁住太过粗暴，连"读任务"也无法并行，任务执行过程本质上是串行的。

于是出现了**共享锁**与**排他锁**：

（1）共享锁（**S**hare Locks，记为S锁），读取数据时加S锁；

（2）排他锁（e**X**clusive Locks，记为X锁），修改数据时加X锁；

共享锁与排他锁的玩法是：

（1）共享锁之间不互斥，简记为：读读可以并行；

（2）排他锁与任何锁互斥，简记为：写读，写写不可以并行；

可以看到，一旦写数据的任务没有完成，数据是不能被其他任务读取的，这对并发度有较大的影响。

画外音：对应到数据库，可以理解为，写事务没有提交，读相关数据的select也会被阻塞。

有没有可能，进一步提高并发呢?

即使写任务没有完成，其他读任务也可能并发，这就引出了数据多版本。

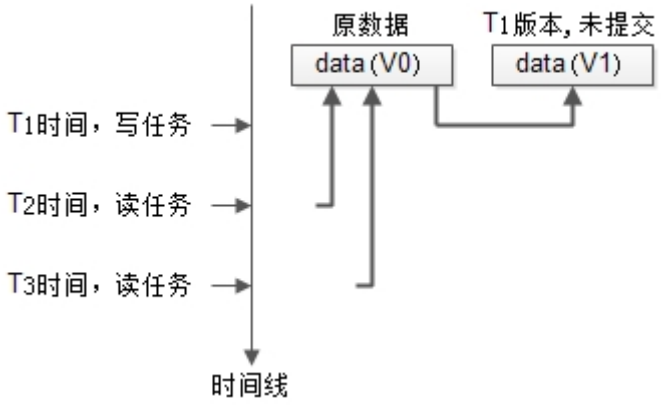
第三节、数据多版本

数据多版本是一种能够进一步提高并发的方法，它的**核心原理**是：

（1）写任务发生时，将数据克隆一份，以版本号区分；

（2）写任务操作新克隆的数据，直至提交；

（3）并发读任务可以继续读取旧版本的数据，不至于阻塞；



如上图：

（1）最开始数据的版本是V0；

（2）T1时刻发起了一个写任务，这是把数据clone了一份，进行修改，版本变为V1，但任务还未完成；

（3）T2时刻并发了一个读任务，依然可以读V0版本的数据；

（4）T3时刻又并发了一个读任务，依然不会阻塞；

可以看到，数据多版本，通过“读取旧版本数据”能够极大提高任务的并发度。

提高并发的演进思路，就在如此：

（1）普通锁，本质是串行执行；

（2）读写锁，可以实现读读并发；

（3）数据多版本，可以实现读写并发；

画外音：这个思路，比整篇文章的其他技术细节更重要，希望大家牢记。

好，对应到InnoDB上，具体是怎么玩的呢？

第四节、redo, undo, 回滚段

在进一步介绍InnoDB如何使用"读取旧版本数据"极大提高任务的并发度之前，有必要先介绍下redo日志，undo日志，回滚段（rollback segment）。

为什么要有redo日志？**

数据库事务提交后，必须将更新后的数据刷到磁盘上，以保证ACID特性。磁盘**随机写**性能较低，如果每次都刷盘，会极大影响数据库的吞吐量。

优化方式是，将修改行为先写到redo日志里（此时变成了**顺序写**），再定期将数据刷到磁盘上，这样能极大提高性能。

*画外音：这里的架构设计方法是，**随机写优化为顺序写**，思路更重要。*

假如某一时刻，数据库崩溃，还没来得及刷盘的数据，在数据库重启后，会重做redo日志里的内容，以保证已提交事务对数据产生的影响都刷到磁盘上。

一句话，redo日志用于保障，已提交事务的ACID特性。

为什么要有undo日志？**

数据库事务未提交时，会将事务修改数据的镜像（即修改前的旧版本）存放到undo日志里，当事务回滚时，或者数据库奔溃时，可以利用undo日志，即旧版本数据，撤销未提交事务对数据库产生的影响。

画外音：更细节的，

*对于**insert操作**，undo日志记录新数据的PK(ROW_ID)，回滚时直接删除；*

*对于**delete/update操作**，undo日志记录旧数据row，回滚时直接恢复；*

他们分别存放在不同的buffer里。

一句话，undo日志用于保障，未提交事务不会对数据库的ACID特性产生影响。

什么是回滚段?

存储undo日志的地方，是回滚段。

undo日志和回滚段和InnoDB的MVCC密切相关，这里举个例子展开说明一下。

栗子：

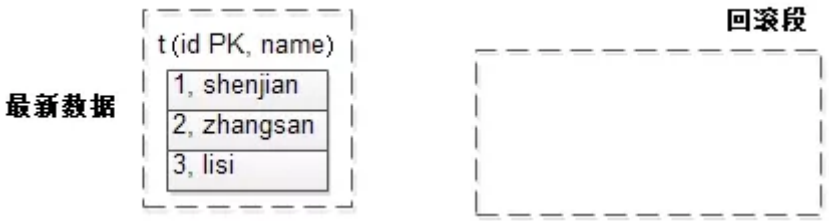
t(id PK, name);

数据为：

1, shenjian

2, zhangsan

3, lisi



此时没有事务未提交，故回滚段是空的。

接着启动了一个事务：

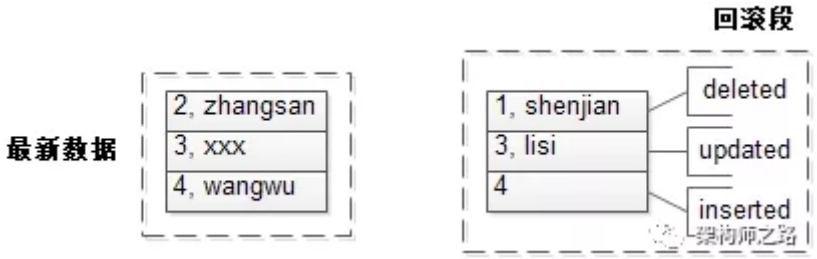
start trx;

delete (1, shenjian);

update set(3, lisi) to (3, xxx);

insert (4, wangwu);

并且事务处于未提交的状态。



可以看到：

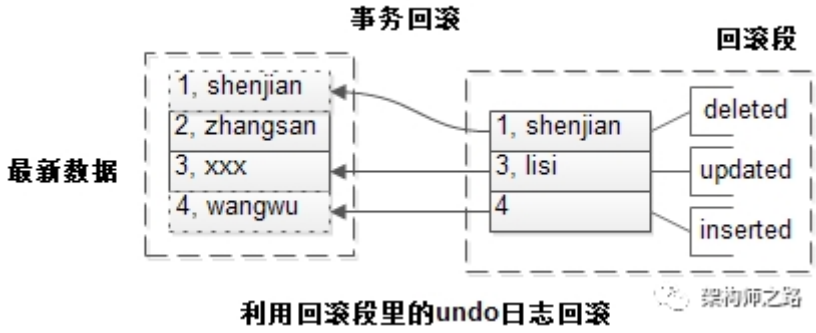
（1）被删除前的(1, shenjian)作为旧版本数据，进入了回滚段；

（2）被修改前的(3, lisi)作为旧版本数据，进入了回滚段；

（3）被插入的数据，PK(4)进入了回滚段；

接下来，假如事务rollback，此时可以通过回滚段里的undo日志回滚。

画外音：假设事务提交，回滚段里的undo日志可以删除。



可以看到：

（1）被删除的旧数据恢复了；

（2）被修改的旧数据也恢复了；

（3）被插入的数据，删除了；

事务回滚成功，一切如故。

第四节、InnoDB是基于多版本并发控制的存储引擎****

InnoDB是高并发互联网场景最为推荐的存储引擎，根本原因，就是其**多版本并发控制**（Multi Version Concurrency Control, MVCC）。行锁，并发，事务回滚等多种特性都和MVCC相关。

MVCC就是通过“读取旧版本数据”来降低并发事务的锁冲突，提高任务的并发度。

核心问题：

旧版本数据存储在哪里？

存储旧版本数据，对MySQL和InnoDB原有架构是否有巨大冲击？**

通过上文undo日志和回滚段的铺垫，这两个问题就非常好回答了：

（1）旧版本数据存储在回滚段里；

（2）对MySQL和InnoDB原有架构体系冲击不大；

InnoDB的内核，会对所有row数据增加三个内部属性：

（1）**DB_TRX_ID**，6字节，记录每一行最近一次修改它的事务ID；

（2）**DB_ROLL_PTR**，7字节，记录指向回滚段undo日志的指针；

（3）**DB_ROW_ID**，6字节，单调递增的行ID；

InnoDB为何能够做到这么高的并发？**

回滚段里的数据，其实是历史数据的快照（snapshot），这些数据是不会被修改，select可以肆无忌惮的并发读取他们。

快照读（Snapshot Read），这种**一致性不加锁的读**（Consistent Nonlocking Read），就是InnoDB并发如此之高的核心原因之一。

这里的**一致性**是指，事务读取到的数据，要么是事务开始前就已经存在的数据（当然，是其他已提交事务产生的），要么是事务自身插入或者修改的数据。

什么样的select是快照读？**

除非显示加锁，普通的select语句都是快照读，例如：

*select * from t where id>2;*

这里的显示加锁，非快照读是指：

*select * from t where id>2* ***lock in share mode****;*

*select * from t where id>2* ***for update****;*

问题来了，这些显示加锁的读，是什么读？会加什么锁？和事务的隔离级别又有什么关系？且听下回分解。

总结

（1）常见并发控制保证数据一致性的方法有**锁**，**数据多版本**；

（2）**普通锁**串行，**读写锁**读读并行，**数据多版本**读写并行；

（3）**redo日志**保证已提交事务的ACID特性，设计思路是，通过顺序写替代随机写，提高并发；

（4）**undo日志**用来回滚未提交的事务，它存储在回滚段里；

（5）InnoDB是基于**MVCC**的存储引擎，它利用了存储在回滚段里的undo日志，即数据的旧版本，提高并发；

（6）InnoDB之所以并发高，快照读不加锁；

（7）InnoDB所有普通select都是快照读；

画外音：本文的知识点均基于MySQL5.6。