Mysql事务 101

1 为什么需要事务

在网上的很多资料里，其实没有很好的解释为什么我们需要事务。其实我们去学习一个东西之前，还是应该了解清楚这个东西为什么有用，硬生生的去记住事务的ACID特性、各种隔离级别个人认为没有太大意义。设想一下，如果没有事务，可能会遇到什么问题，假设你要对x和y两个值进行修改，在修改x完成之后，由于硬件、软件或者网络问题，修改y失败，这时候就出现了"部分失败"的现象，x修改成功，y修改失败，这个时候需要你自己在应用代码里去处理，你可以重试修改y，也可以把x设置成之前的值（回滚），不管你怎么做，这些由于底层系统的各种错误导致的问题，都需要你自己写应用代码去处理，而如果有了事务，你完全没必要关心这些底层的问题，只要提交成功了，所有的修改都是成功的，如果有失败的，事务会自动回滚回之前的状态；另外，在并发修改的场景中，如果没有事务，你需要自己去实现各种加锁的逻辑，繁琐而且容易出错，而如果有了事务，你可以通过选择事务的一个隔离级别，来假装某些并发问题不会出现，因为数据库已经帮你处理好了。总之，事务是数据库为我们提供的一层抽象，让我们假装底层的故障和某些并发问题并不存在，从而更加舒服的编写业务代码

2 什么是事务

众所周知，事务有着ACID属性，分别是原子性（atomicity），一致性（consistency），隔离性（isolation）和持久性（durability），我们分别展开说一下

2.1 原子性

首先可能有人会混淆这里的原子性和多线程编程中的原子操作，多线程编程中的原子操作是指如果一个线程做了一个原子操作，其他线程无法看到这个操作的中间状态，这是一个有关并发的概念。而事务的原子性指的是一个事务中的多个操作，要么全部成功，要么全部失败，不会出现部分成功、部分失败的情况。

2.2 一致性

如果说原子性是一个有点容易混淆的概念，一致性这个概念就更加模糊了，可能很多人看到这个词都不知道他在说啥。一致性是什么，我们看一下《数据库系统概论》这本书给的定义：

> （一致性是指）事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态

那什么叫一致性状态，其实就是你对数据库里的数据有一些约束，例如主键必须是唯一的、外键约束这些（还记得数据库的完整性约束吗），当然，更多的是业务层面的一致性约束，比如在转账场景中，我们要求事务执行后所有人的钱总和没有改变。数据库可以帮我们保证外键约束等数据库层面的一致性，但是对我们业务层面的一致性是一无所知的，比如你可以给每个人的钱都加100块，数据库并不会阻止你，这时你就轻松的违反了业务层面的一致性。所以我们可以发现，一致性对我们来说是一个有点无关痛痒的属性，我们其实是通过事务提供的原子性和隔离性来保证事务的一致性，甚至你可以认为一致性不属于事务的属性，也有人说一致性之所以存在，只是为了让ACID这个缩略词更加顺口而已

2.3 隔离性

如果多个事务同时操作了相同的数据，那么就会有并发的问题出现，比如说多个事务同时给一个计数器（counter）加1，假设counter初始值为0，那么可能会出现这样的情况：

时间	事务A	事务B
T1	读计数器 counter = 0	
T2		读计数器 counter = 0
T3	写计数器 counter = 1	
T4		写计数器 couter = 1

我们做了两次加1操作，结果本应是2，但是最终可能会是1。当然，还会有其他的并发问题，隔离性就是为了屏蔽掉一些并发问题的处理，让我们编写应用代码更加简单。我们再来看一下《数据库系统概论》给隔离性的定义：

> 一个事务的执行不能被其他事务干扰。即一个事务的内部操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能相互干扰

课本上的定义是根据"可串行化"这个隔离级别来表述隔离性，就是说你可以认为事务之间完全隔离，就好像并发的事务是顺序执行的。但是，我们实际用的时候，为了更好的并发性能，基本不会把事务完全隔离，所以就有了隔离级别的概念，sql 92标准定义了四种隔离级别：未提交读、提交读、可重复读、可串行化，大家一般会使用较弱的隔离级别，例如"可重复读"。关于各种隔离级别，我们放到第三部分和第四部分再说

2.4 持久性

持久性是指一旦事务提交，即使系统崩溃了，事务执行的结果也不会丢失。为了实现持久性，数据库会把数据写入非易失存储，比如磁盘。当然持久性也是有个度的，例如假设保存数据的磁盘都坏了，那持久性显然无法保证

3 并发问题

首先，事务既然提供了隔离级别的抽象，那么含义就是在使用的时候，不需要自己去加锁处理某一类的并发问题，所以很多资料在通过自己手动加锁做了一些实验之后，就得出Mysql的可重复读隔离级别能够防止丢失更新、幻读等结论显然是不正确的，至于Mysql能提供什么保证，我们放到第五部分再说

我们前面提到，为了更好的并发性能，我们搞出了各种弱隔离级别，那么隔离级别是怎么定义的呢？隔离级别是通过可能遇到的并发问题（异象）来定义的，选定一个隔离级别后，就不会出现某一类并发问题，那么我们就来看看会有哪些并发问题，在每一小节，我们会先讲讲这个并发问题是什么，然后讨论阻止他的隔离级别，最后说说实现这个隔离级别的方法，这里我们只讨论加锁的实现，其他实现我们放到第四章来讲，所以我们先简单说一下锁

- S锁和X锁大家应该都很熟悉，S锁即共享锁，X锁即互斥锁

- 根据锁持有的时间，我们把锁分为Short Duration Lock和Long Duration Lock，本文就简称为短锁和长锁，短锁即语句执行前加锁，执行完成后就释放；长锁则是语句执行前加锁，而到事务提交后才释放
- 另外根据锁的作用对象，我们把锁分为记录锁（record lock）和谓词锁（predicate lock），谓词锁顾名思义锁住了一个谓词，而不是具体的数据记录，比如select * from table where id > 10，如果加谓词锁，就锁住了10到无限大这个范围，不管表里是否真的存在大于10的记录

3.1 脏写（Dirty Write）

一个事务对数据进行写操作之后，还没有提交，被另一个事务对相同数据的写操作覆盖（你可能会看到有的资料称之为"第一类丢失更新"）

举个例子：（x初始值为0）

时间	事务A	事务B
T1	begin	begin
T2	写x x = 1	
T3		写x x = 2
T4	commit	
T5		commit

这里事务A将x写为1之后还没有提交，就被事务B覆盖为2。

3.1.1 问题

脏写会导致什么问题呢?

第一个问题是无法回滚，假设在T4时刻事务A要回滚，这个时候x的值已经变成了2，如果把x回滚为事务A修改之前的值，也就是0，那么事务B的修改就丢失了；如果不回滚，那么当T5时刻，事务B也要回滚时，你还是不能回滚x的值，因为事务B修改之前，x的值是1，由于事务A回滚，这个值已经变成了脏数据。这就导致事务没办法回滚，影响了事务的原子性

第二个问题是影响一致性，假如说我们同时对x和y进行修改，要求x和y始终是相等的，看下面的例子

初始值 x=y=1

时间	事务A	事务B
T1	begin	begin
T2	写x x = 2	
T3		写x x = 3
T4		写y y = 3
T5	写y y = 2	
T6	commit	commit

可以看到，最终x变成了3，y变成了2，违反了一致性

3.1.2 隔离级别

由于脏写导致不能回滚，严重影响原子性，所以不管是什么隔离级别，都要阻止这种问题，因此可以认为最弱的隔离级别"未提交读"需要阻止脏写

3.1.3 实现

那怎么防止脏写呢，很简单，就是加锁，一般会在更新之前加行级锁（X锁），那什么时候释放锁呢，更新操作执行完之后释放锁显然不行，必须等到事务提交之后再释放锁，这样才不会出现脏写的情况，即写操作加长锁（X锁）

3.2 脏读（Dirty Read）

一个事务对数据进行写操作之后，还没有提交，被另一个事务读取

3.2.1 问题

脏读会导致什么问题呢？我们看两个例子

第一个：

x初始值为100

时间	事务A	事务B
T1	begin	begin
T2	写x x = 200	
T3		读x x = 200

时间	事务A	事务B
T4	rollback	...

可以看到由于事务A回滚，事务B读到的x值变成了脏数据

那如果事务A不回滚，事务B读到的不就不是脏数据了吗?

其实同样可能有问题，我们看第二个例子:

假设x=50 y=50，x要给y转账40，那我们的一致性要求就是x+y在事务执行后仍然为100

时间	事务A	事务B
T1	begin	begin
T2	写x x = 10	
T3		读x x = 10
T4		读y y = 50
T5		commit
T6	写y y = 90	
T7	commit	

可以看到事务B读到的结果是x+y=60，违反了我们要求的一致性

3.1.2 隔离级别

SQL-92定义了"提交读"隔离级别来阻止脏读，不过SQL-92只提到了第一种情况，而其实不管有没有回滚，只要读到了其他事务未提交的数据，都应该认为是脏读，都可能会出现问题

3.1.3 实现

提交读如何实现呢？我们为了防止脏写，已经对写操作加了长锁，那么在此基础上，只要给读操作加短锁（S锁）就能解决脏读的问题，即读之前申请锁，读完后立即释放，注意，这里不仅要给数据记录加短锁，还要加谓词锁，为什么呢，试想假如只加记录锁，如果我们做了一个范围查询，而在查询过程中，正好另外一个事务在这个范围插入了一条数据，我们的范围查询仍然能够读到，即读到了其他事务未提交的数据，因此还需要加谓词锁（短锁，S锁）。总之，实现提交读，需要写操作加长锁，读操作加短锁（记录锁和谓词锁）

3.3 不可重复读

不可重复读（Non-Repeatable Read）也叫Fuzzy Read，指一个事务对数据进行读操作后，该数据被另一个事务修改，再次读取数据和原来不一致（其实不读第二次也可能会有问题）

3.3.1 问题

我们还是看两个例子：

第一个：

x初始值为1

时间	事务A	事务B
T1	begin	begin

时间	事务A	事务B
T2	读x x = 1	
T3		写x x = 2
T4		commit
T5	读x x = 2	
T6	commit	

可以看到事务A第二次读取x的值发生了变化，影响了一致性

那么如果我没有对相同数据做第二次读取呢?

我们看第二个例子:

x初始值为50，y初始值为50，x给y转账40，我们的一致性要求时事务执行后x和y的总和不变

时间	事务A	事务B
T1	begin	begin
T2	读x x = 50	

时间	事务A	事务B
T3		写x x = 10
T4		写y y = 90
T5		commit;
T6	读y y = 90	
T7	commit	

可以发现，事务A没有对任何数据读第二次，但是在事务A看来，x+y=140，而不是100，违反了一致性

3.3.2 隔离级别

SQL-92定义了“可重复读”隔离级别来阻止不可重复读的问题，但是它只提到了第一种情况，但是从第二个例子我们可以发现，不管有没有做第二次读取，其实都可能会有问题，因此要想阻止不可重复读，事务读完数据后，就要阻止其他事务对该数据的写操作

3.3.3 实现

在“提交读”中，我们已经对写操作加了长锁，对读操作加了短锁（记录锁，谓词锁），为了阻止不可重复读的问题，需要给读操作中的记录锁也加长锁（S锁），因此“可重复读”隔离级别的实现就是读操作中记录锁加长锁（S锁），谓词锁加短锁（S锁），写操作加长锁（X锁）

这里在记录锁的角度来看，我们其实已经在做两阶段锁（Two Phase Locking: 2PL）了。我们简单讨论一下两阶段锁：

顾名思义，两阶段锁一定有两个阶段：

- Expanding phase（也叫Growing phase），即加锁阶段，这个阶段可以加锁，但是不能释放锁
- Shrinking phase（也叫Contracting phase），即解锁阶段，这个阶段可以解锁，但是不能再加锁了

两阶段锁有几种变体，比较常见的就是两种：

- 保守两阶段锁（Conservative two-phase locking），就是在开始之前一次性把要加的锁加上，也就是一些资料说的“一次封锁法”，可以防止死锁
- 严格两阶段锁（Strict two-phase locking），X锁在提交之后才能释放，S锁可以在解锁阶段释放

我们这里，包括在很多资料提到的两阶段锁，其实是指严格两阶段锁

3.4 幻读

幻读是指一个事务通过一些条件进行了读操作，比如select * from table where id > 1 and id < 10，然后另一个事务的写操作改变了匹配该条件的数据（可能是插入了新数据，可能是删除了匹配条件的数据，也可能是通过更新操作让其他数据也变得匹配该条件）

3.4.1 问题

同样看两个例子：

假设学生表中有a，b，c三个学生

时间	事务A	事务B
T1	begin	begin
T2	读所有学生列表 学生列表为a，b，c	
T3		添加学生d 学生列表为a，b，c，d

时间	事务A	事务B
T4		commit;
T5	读所有学生列表 学生列表为a，b，c，d	
T6	commit	

可以看到，事务A第二次读取所有学生列表，多了一个学生出来，影响了一致性

那么，重新问一下在不可重复读中问过的问题，如果我不做第二次读取呢?

答案是同样可能有问题，我们看第二个例子:

还是这个学生表，有a，b，c三个学生，同时为了避免直接计数的性能问题，我们还有一个count记录学生的总数，count初始值为3

时间	事务A	事务B
T1	begin	begin
T2	读所有学生列表 学生列表为a，b，c，因此学生总数为3	
T3		添加学生d 学生列表为a，b，c，d
T4		写count count = 4
T5		commit;

时间	事务A	事务B
T6	读count count = 4 得到学生总数是4	
T7	commit	

这次我们没有读取两次所有学生列表，但是可以看到两个有关联的数据发生了不一致，明明读学生列表后我们计算出的总数是3，可是直接读count得到的却是4，违反了一致性

3.4.2 隔离级别

SQL-92定义了“可串行化”隔离级别来阻止幻读的问题，不过对于幻读问题只提及了第一种情况，而其实不管有没有第二次读取，只要其他事务的写导致读取的结果集发生变化，都可能会发生一致性的问题

3.4.3 实现

在“可重复读”隔离级别中，我们已经给读操作加了记录锁（长锁）和谓词锁（短锁），为了防止幻读，谓词锁加短锁已经不行了，我们需要把谓词锁也变成长锁。因此可串行化隔离级别的实现就是读操作加长锁（记录锁，谓词锁），写操作加长锁，也就是通过两阶段锁来实现可串行化。

3.5 丢失更新（Lost Update）

因为SQL-92对异象的定义不够完整，后面要提到的三种异象可能稍微陌生一些

丢失更新是指一个事务的写被另一个已提交事务覆盖（有些资料把它称为第二类丢失更新）

3.5.1 问题

我们看一个例子：

counter初始值为1，两个事务分别给counter值加1，counter最后的值应该变成3

时间	事务A	事务B
T1	begin	begin
T2	读couter counter = 1	读couter counter = 1
T3	写counter counter = 2	
T4	commit	
T5		写counter counter = 2
T6		commit

我们发现事务B提交之后counter值是2，也就是说即使事务A已经提交了，它对counter的更新却"丢失"了

3.5.2 隔离级别

由于SQL-92没有提及这种异象，所以对于哪种隔离级别应该阻止丢失更新没有权威的定义，不过我们可以看到上面会出现丢失更新的问题，是因为事务B读取counter后被事务A修改，这是上面的的"可重复读"隔离级别加锁实现所阻止的，因此我们对于可重复读的加锁实现能够阻止"丢失更新"的发生，上面的例子中，由于对读操作加了长锁，所以两个事务的写操作会互相等待对方的读锁释放，形成死锁，如果有死锁检测机制，事务B会自动回滚，不会出现丢失更新的情况

3.6 Read Skew

最后两个异象Read Skew和Write Skew都是违反了数据原有的一致性约束

Read Skew即读违反一致性约束，原本多个数据存在一致性的约束，读取发现违反了该一致性

3.6.1 问题

我们直接用不可重复读中的第二个例子就好：

x初始值为50，y初始值为50，x给y转账40，我们的一致性要求时事务执行后x和y的总和不变

时间	事务A	事务B
T1	begin	begin
T2	读x x = 50	
T3		写x x = 10
T4		写y y = 90
T5		commit;
T6	读y y = 90	
T7	commit	

事务A发现x+y变成140了，这就出现了Read Skew，你可以把Read Skew当成不可重复读的一种情况

3.6.2 隔离级别

SQL-92同样没有提及这种异象，由于Read Skew可以视为不可重复读的一种情况，所以"可重复读"隔离级别应该阻止Read Skew（我们对于可重复读的加锁实现能够阻止"Read Skew"的发生，上面的例子中，事务B的写操作会被事务A的读锁阻塞，因此事务A会读到x=y=50，不会出现Read Skew）

3.7 Write Skew

write skew即写违反一致性约束，通常发生在根据读取的结果进行写操作时，并发事务的写操作导致最终结果违反了一致性约束，可能不好理解，我们看个例子

3.7.1 问题

第一个问题：

假设x和y是一个人的两个信用卡账户，我们要求x + y不能小于0，而x或者y可以小于0，就是说你的一张信用卡可以是负的，但是全部加起来不能也是负的

下面事务A和事务B是两次并发的扣款，x初始值为20，y初始值为20

时间	事务A	事务B
T1	begin	begin
T2	读x x = 20 y = 20	读x x = 20 y = 20
T3	发现还有40块钱，扣款30	

时间	事务A	事务B
T4	写x x = -10	
T5		发现还有40块钱，扣款30
T6		写y y = -10
T7	commit	
T8		commit

我们发现两个事务提交之后，x+y变成了-20，违反了一致性约束

上面这个问题是严格意义上的Write Skew，另外还有由于幻读产生的Write Skew

问题2：

假设我们要做一个注册用户的功能，要求用户名唯一，并且没有给用户名加唯一索引，也就是说唯一性我们自己来保证

用户表已有用户名a，b，c，两个用户同时注册用户名d

时间	事务A	事务B
T1	begin	begin
T2	读所有用户名 当前用户名：a，b，c	读所有用户名 当前用户名：a，b，c

时间	事务A	事务B
T3	发现没有d	
T4	插入用户名d	
T5	commit	
T6		发现没有d
T7		插入用户名d
T8		commit

我们发现，两个事务提交之后，用户名d有了两个，违反了唯一性约束

这个问题由于是幻读引发的，所以有人把它归类在Write Skew里，也有人把它归类在幻读里，你可以按照自己的理解来分类

3.7.2 隔离级别

SQL-92也没有提及Write Skew，我们上面提到了两个问题，一个是严格意义上的Write Skew，一个是幻读引发的Write Skew，如果是严格意义上的Write Skew，我们上面的"可重复读"隔离级别加锁实现可以阻止（写操作会被读锁阻塞）；而由于幻读引发的Write Skew，本质上已经是幻读问题，所以只有"可串行化"隔离级别能够阻止（上面的例子中，由于谓词锁的存在，后面的插入操作被阻塞）

3.8 隔离级别汇总

我们最后对各种隔离级别的加锁实现汇总一下：

隔离级别	读操作（S锁）	写操作（X锁）
未提交读	不需要	加长锁

隔离级别	读操作（S锁）	写操作（X锁）
提交读	记录锁加短锁 谓词锁加短锁	加长锁
可重复读	记录锁加长锁 谓词锁加短锁	加长锁
可串行化	记录锁加长锁 谓词锁加长锁	加长锁

另外对于基于锁实现的隔离级别，我们根据其避免的并发问题汇总一下

隔离级别	脏写	脏读	不可重复读	幻读	丢失更新	Read Skew	Writ
未提交读	不会	会	会	会	会	会	会
提交读	不会	不会	会	会	会	会	会
可重复读	不会	不会	不会	会	不会	不会	不会
可串行化	不会	不会	不会	不会	不会	不会	不会

4 其他隔离级别

在上面讨论各种异象的过程中，我们也引入了一些隔离级别，包括：

- 未提交读
- 提交读

- 可重复读
- 可串行化

我们也探讨了相关隔离级别的基于锁的实现，你会发现除了未提交读，我们都需要对读加锁了，这可能会带来性能问题，一个执行时间稍长的写事务，会阻塞大量的读操作。因此，为了提高性能，很多数据库实现都是采用数据多版本的方式，即保留旧版本的数据，可以做到读操作不必加锁，因此读不会阻塞写，写也不会阻塞读，可以获得很好的性能。因此就出现了另外一种隔离级别——快照隔离（Snapshot Isolation），由于SQL-92标准制定时，快照隔离还没有出现，所以快照隔离没有出现在标准中，一些实现快照隔离的厂商也是按照可重复读来宣传自己的数据库产品。当然，除了快照隔离也有其他的隔离级别实现（例如Cursor Stability 游标稳定），我们不会在这里讨论，感兴趣的同学可以自己了解

4.1 快照隔离

快照隔离是指每个事务启动时，数据库就为这个事务提供了这时数据库的状态，即快照（好像把数据库此时的数据都照下来了一样），后续其他事务对数据的新增、修改、删除操作，这个事务都看不到，它始终只能看到自己的一致性快照

4.2 快照隔离实现

快照隔离怎么实现呢，对于写操作还是用长锁来防止脏写的问题，对于读操作，主要思想就是维护多版本的数据，也就是所谓的MVCC（multi-version concurrency control），MVCC不止是用来实现快照隔离这个级别，很多数据库也用它来实现"提交读"隔离级别，区别于快照隔离在事务开始时得到一个一致性快照，在"提交读"隔离级别，每个语句执行时，都会有一个快照。我们这里主要关注MVCC实现"快照隔离"的方式。

MVCC的实现方式主要有两种：

- 维护多版本数据，比较有代表性的是Postgresql
- 维护回滚日志（undo log），比较有代表性的是Mysql InnoDB

我们后面会简单介绍一下这两种方式的实现思想，不过我们不会涉及具体的数据库实现

4.2.1 维护多版本数据

这种方式是实实在在的保留了多个版本的数据，例如假如我有这样一行数据：

id	name	age	gender	version
1	zhangsan	10	male	xxx

如果我把年龄改为20，表中会添加一个不同版本的数据（实际的存储结构可能是B+树，不过为了简单，我们把数据的存储结构简化为一个表格来描述）

id	name	age	gender	version
1	zhangsan	10	male	xxx
1	zhangsan	20	male	yyy

也就是说，即使其他列的值没有变化，也会原样复制一份

那么，怎么实现MVCC呢？

首先，事务开始时数据库会分配一个事务id，我们这里记作txid，数据库保证这个id是单调递增的（我们这里不考虑整数回绕的情况）

另外，数据库会在每行数据添加两个隐藏字段：

- create_by 表示创建这行数据的事务id
- delete_by 表示删除这行数据的事务id

我们分别看一下插入、更新和删除的过程：（使用用户表做例子）

4.2.1.1 插入

假设当前事务id为3，插入一个叫liming的用户

id	name	age	gender	create_by	delete_by
1	liming	10	male	3	null

该数据的create_by为3，delete_by为null

4.2.1.2 更新

更新操作可以转换为：删除原数据+插入新数据

假设事务id为4的事务，将liming的年龄更新为20

id	name	age	gender	create_by	delete_by
1	liming	10	male	3	4
1	liming	20	male	4	null

4.2.1.3 删除

假设事务id为5的事务，将liming删除

id	name	age	gender	create_by	delete_by
1	liming	10	male	3	4
1	liming	20	male	4	5

如上将delete_by修改为5

4.2.1.4 可见性规则

对于当前事务能够"看到"哪些数据，我们用可见性规则来定义

在事务开始时，数据库会获取当前活跃（未提交的）的事务id列表，以及当前分配的最大事务id

这个事务能看到哪些数据遵循以下规则：

- 如果对数据做更改的事务id在活跃事务id列表中，那么这个更改不可见
- 如果对数据做更改的事务id大于当前分配的最大事务id，说明是后续的事务，更改不可见
- 如果对数据做更改的事务是回滚状态，更改不可见

上面我们说的更改包括创建和删除（更新可以转化为删除+创建），"创建"不可见意味着当前事务看不到其他事务新创建的数据，"删除"不可见意味着当前事务仍然能看到其他事务已删除的数据

通过这样的可见性规则我们可以保证事务永远从一个"一致性快照"中读取数据

4.2.2 维护回滚日志（undo log）

这种方式保留了数据的回滚日志，而非所有版本的完整数据，需要查询旧版本数据时，通过在最新数据上应用回滚日志中的修改，构造出历史版本的完整数据，主要思想还是和第一种方式一样，只是采用了另外一种实现方式，其实理解了第一种方式也就理解了MVCC，因此这里我们只简单介绍维护回滚日志的方式

例如这样一行数据

id	name	age	gender	version
1	zhangsan	10	male	xxx

把年龄修改为20，则直接在数据中修改

id	name	age	gender	version
1	zhangsan	20	male	xxx

同时在回滚日志中会记录类似“把age从20改回10”的回滚操作

这种方式怎么实现MVCC呢，和上面一样，事务开始时数据库会分配一个事务id，我们这里记作txid，数据库保证这个id是单调递增的

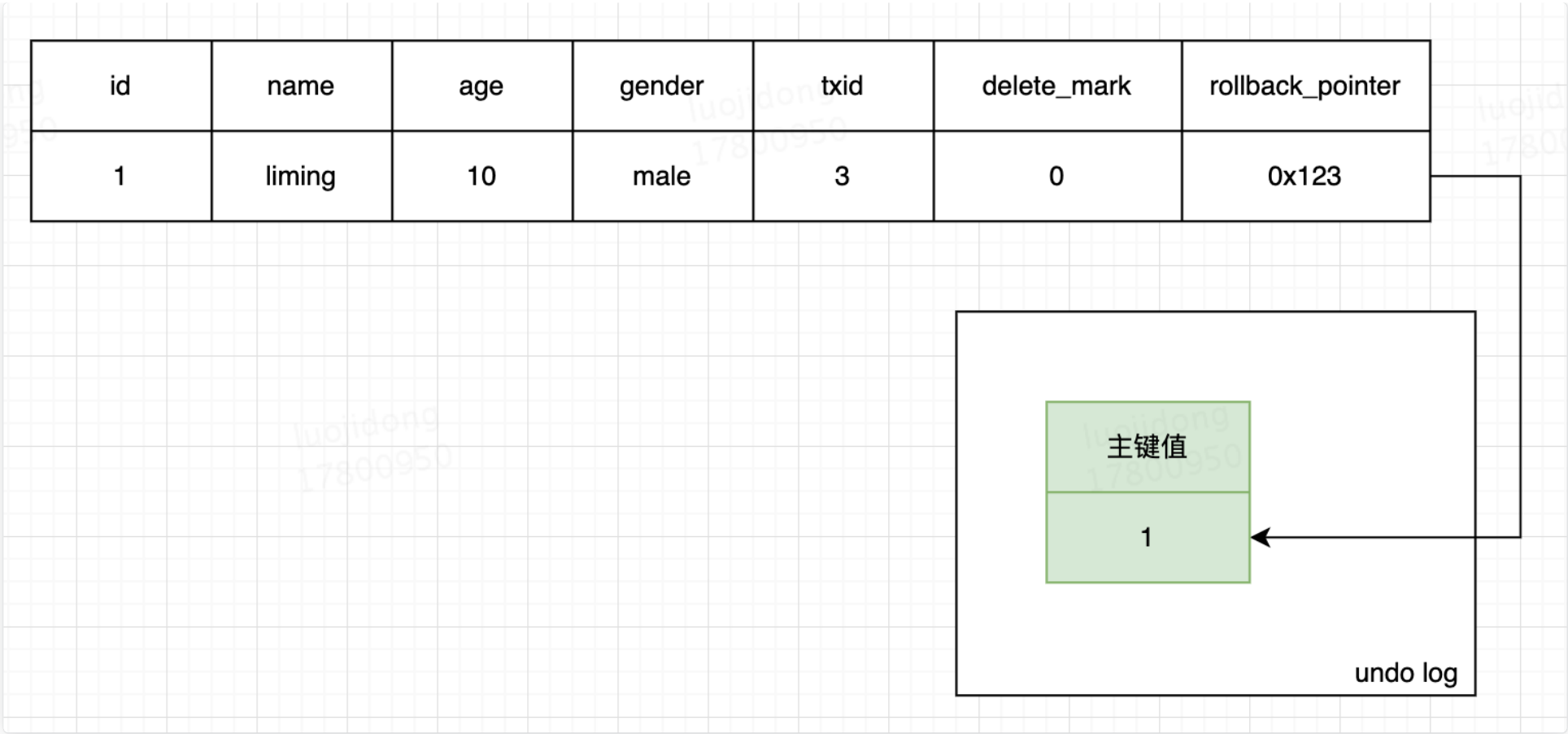
类似上面的create_by和delete_by，数据中也会有一些隐藏字段（我们这里只讨论和MVCC相关的隐藏字段）

- txid，创建该数据的事务id
- rollback_pointer，回滚指针，指向对应的回滚日志记录
- delete_mark，删除标记，标记数据是否删除（我们后面用1来表示已删除，0来表示未删除）

同样，我们看一下插入、更新和删除的过程

4.2.1.1 插入

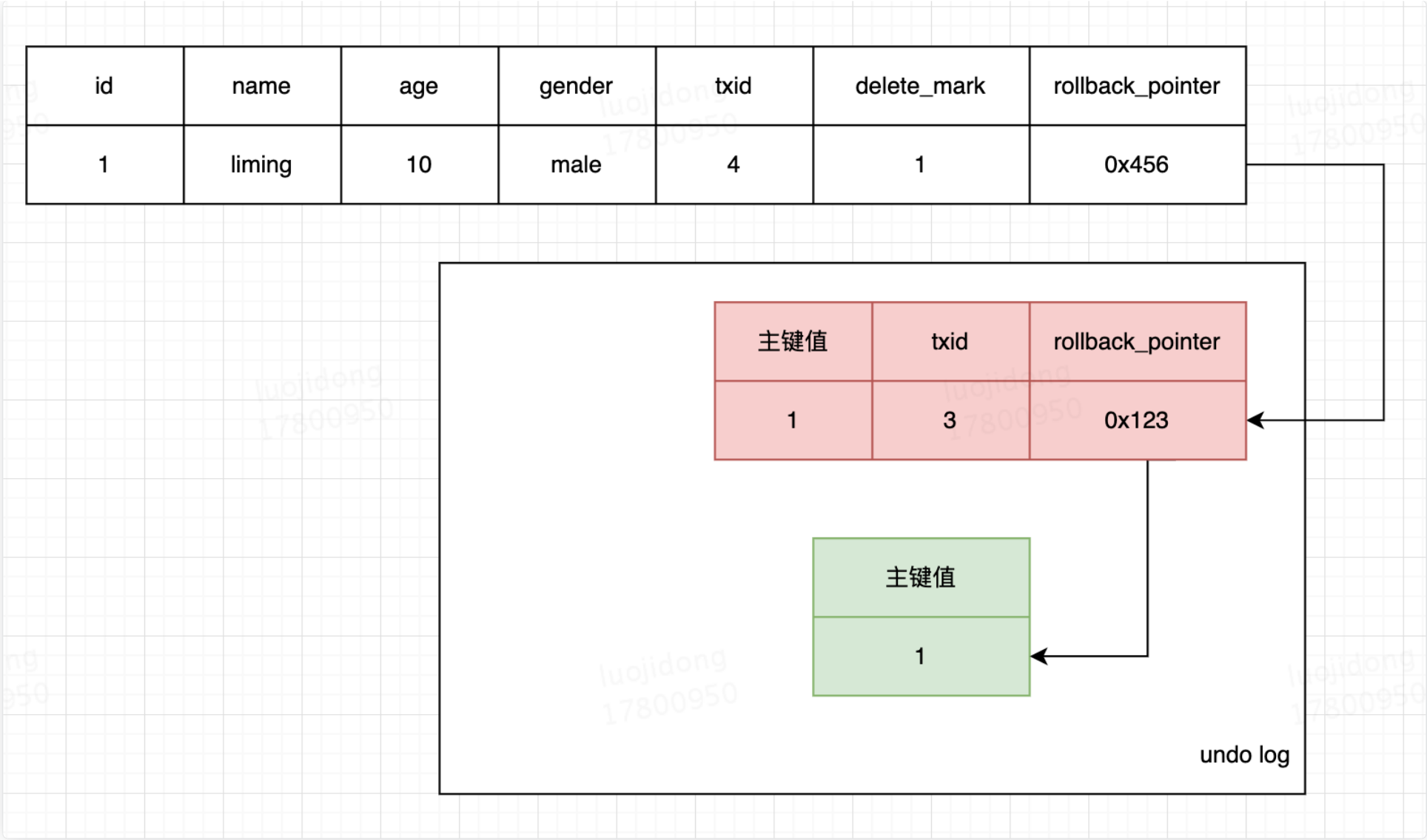
假设当前事务id为3，插入一个叫liming的用户（下面用绿色表示插入对应的回滚日志）



如图，事务id设置为3，删除标记为0，同时在回滚日志中记录该数据的主键值，我们这里主键是id，因此记录1就好，并且将回滚指针指向该回滚日志，这里记录主键值是为了回滚时通过主键值删除相关数据和索引

4.2.1.2 删除

假设事务4要删除liming这条记录（下面用红色表示删除对应的回滚日志）



我们会把liming这条记录的delete_mark设置为1，同时在回滚日志中记录删除前的事务id、回滚指针以及主键值

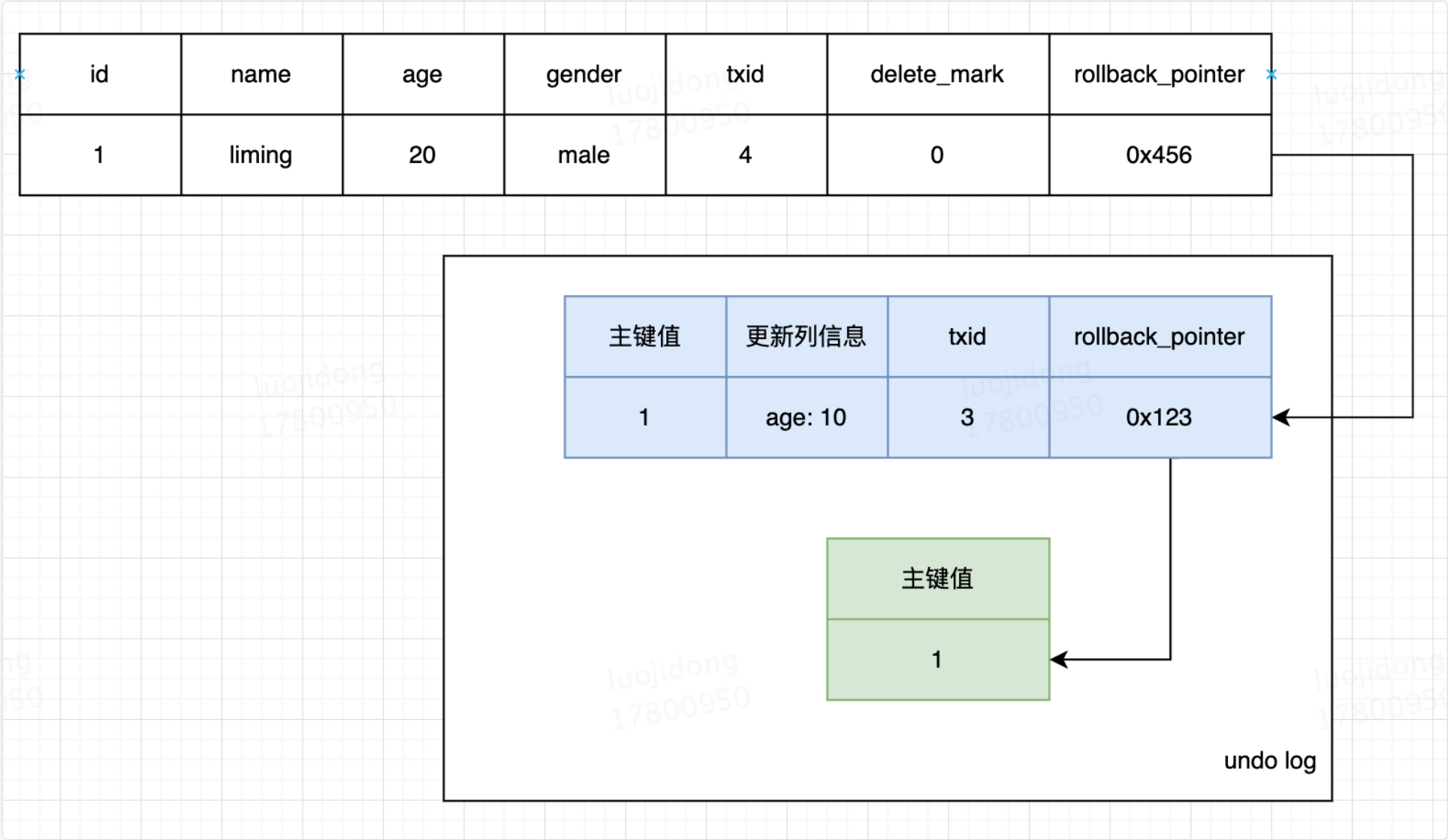
4.2.1.3 更新

更新要分为不更新主键和更新主键两种情况（我们这里假设主键是id）

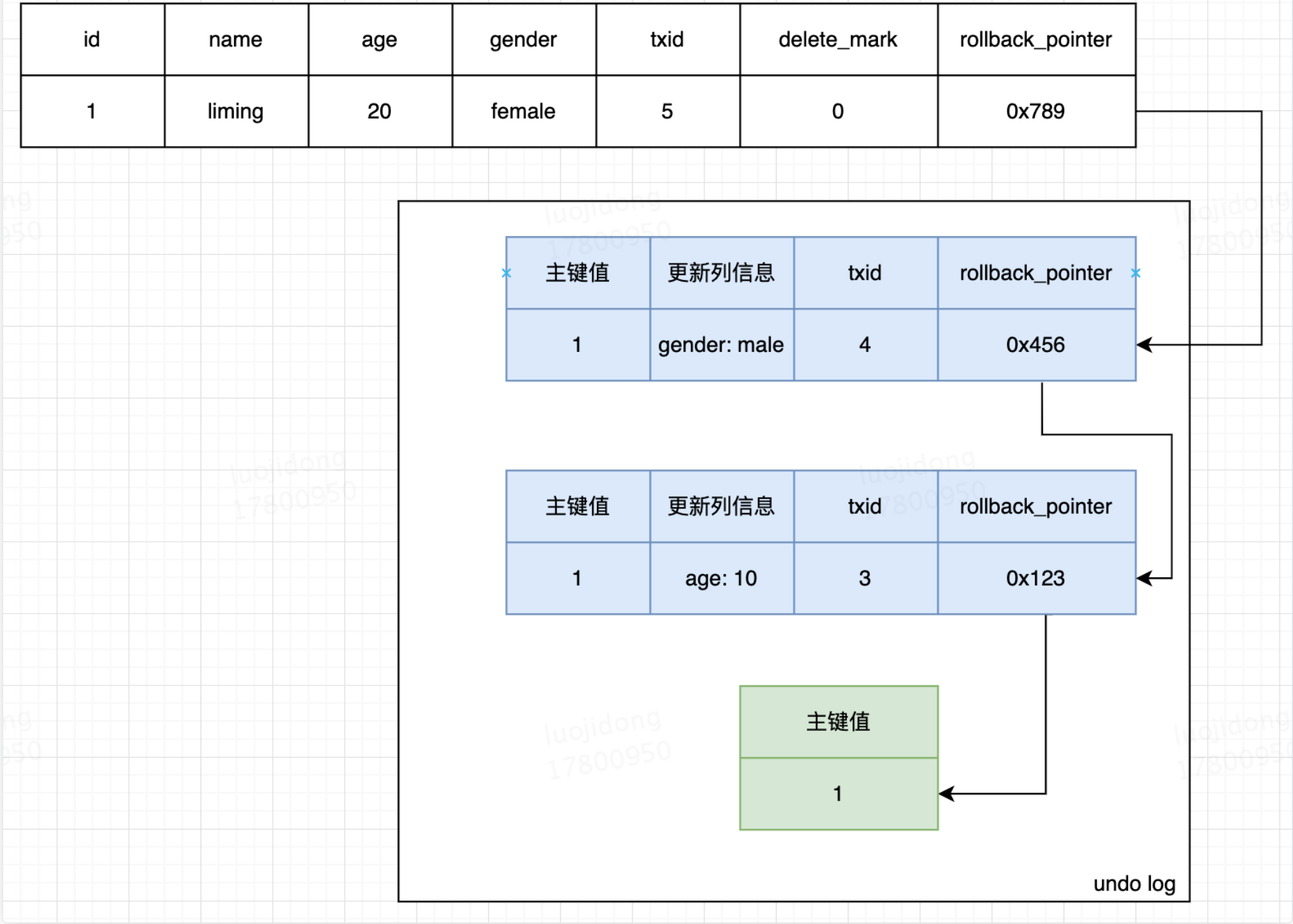
4.2.1.3.1 不更新主键

首先看不更新主键的情况：

假设id为4的事务将之前插入的liming的age更新为20（下面用蓝色表示更新对应的回滚日志）



我们会把原来的age直接更新成20，并且txid改为4，同时在回滚日志中记录更新列的信息，这里是age: 10，表示更新前age的旧版本数据是10，另外我们也记录了原来的事务id和回滚指针，最终回滚日志中的数据会通过回滚指针形成一个链表，从而查找旧版本数据，比如如果事务id为5的事务接着把gender更新为female：


id
name
age
gender
txid
delete_mark
rollback_pointer
1
liming
20
female
5
0
0x789
主键值
更新列信息
txid
rollback_pointer
1
gender: male
4
0x456
主键值
更新列信息
txid
rollback_pointer
1
age: 10
3
0x123
主键值
1
undo log

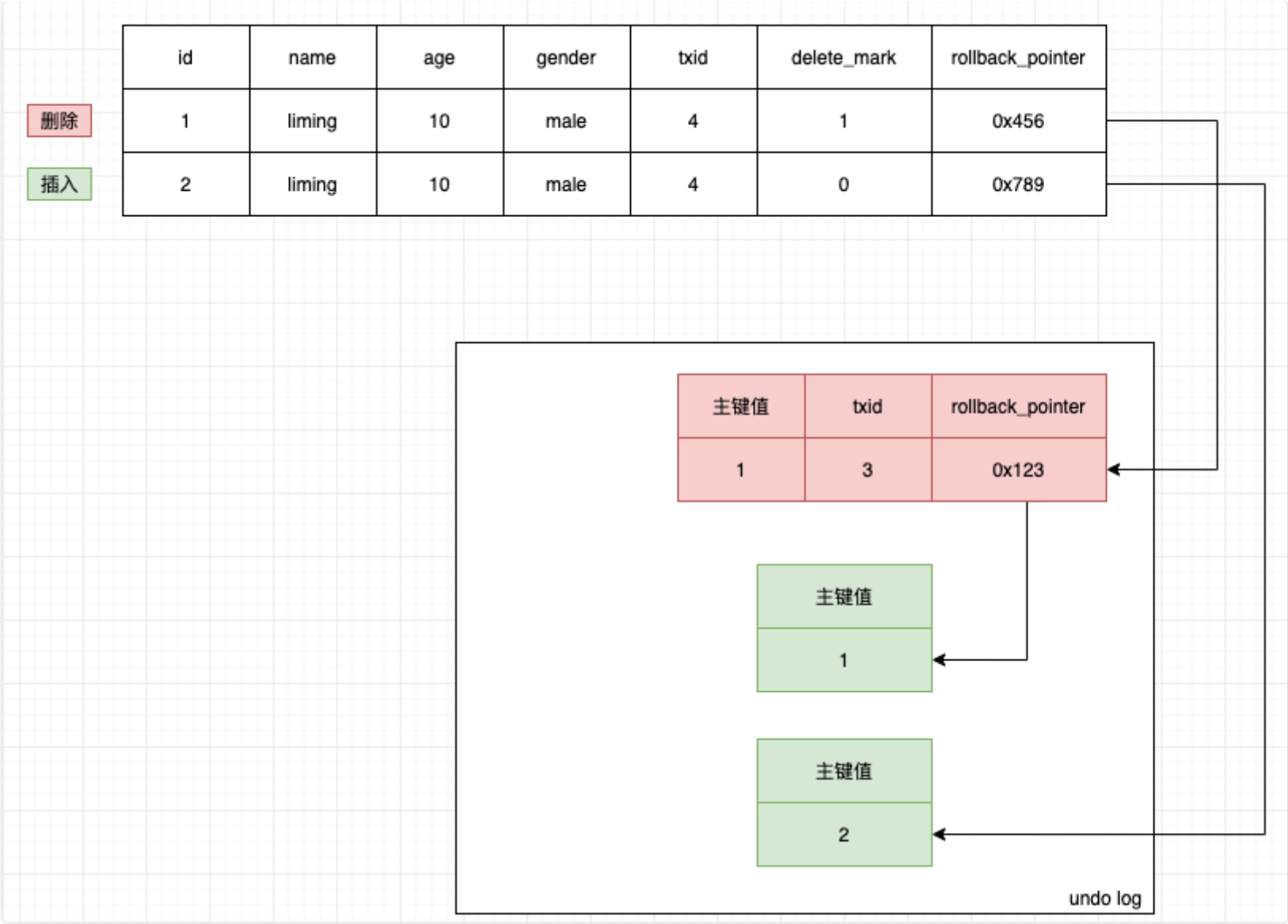
4.2.1.3.2 更新主键

接下来看看更新主键的情况：

更新主键时，和第一种实现MVCC的方式类似，转换为删除+插入

为什么这个时候和不更新主键不一样呢，是因为更新主键时，数据的位置已经发生变化了，比如数据存储的结构是B+树，如果主键更新了，那么数据在B+树中的位置肯定会变化，如果还在旧版本的数据上直接修改主键，那么查找的时候是找不到的（因为是根据主键值做查找），所以这个时候要转换为删除+插入

例如事务id为4的事务，将之前我们插入的liming的id更新为2



4.2.1.4 可见性规则

这里的可见性规则其实和MVCC的第一种实现方式是类似的

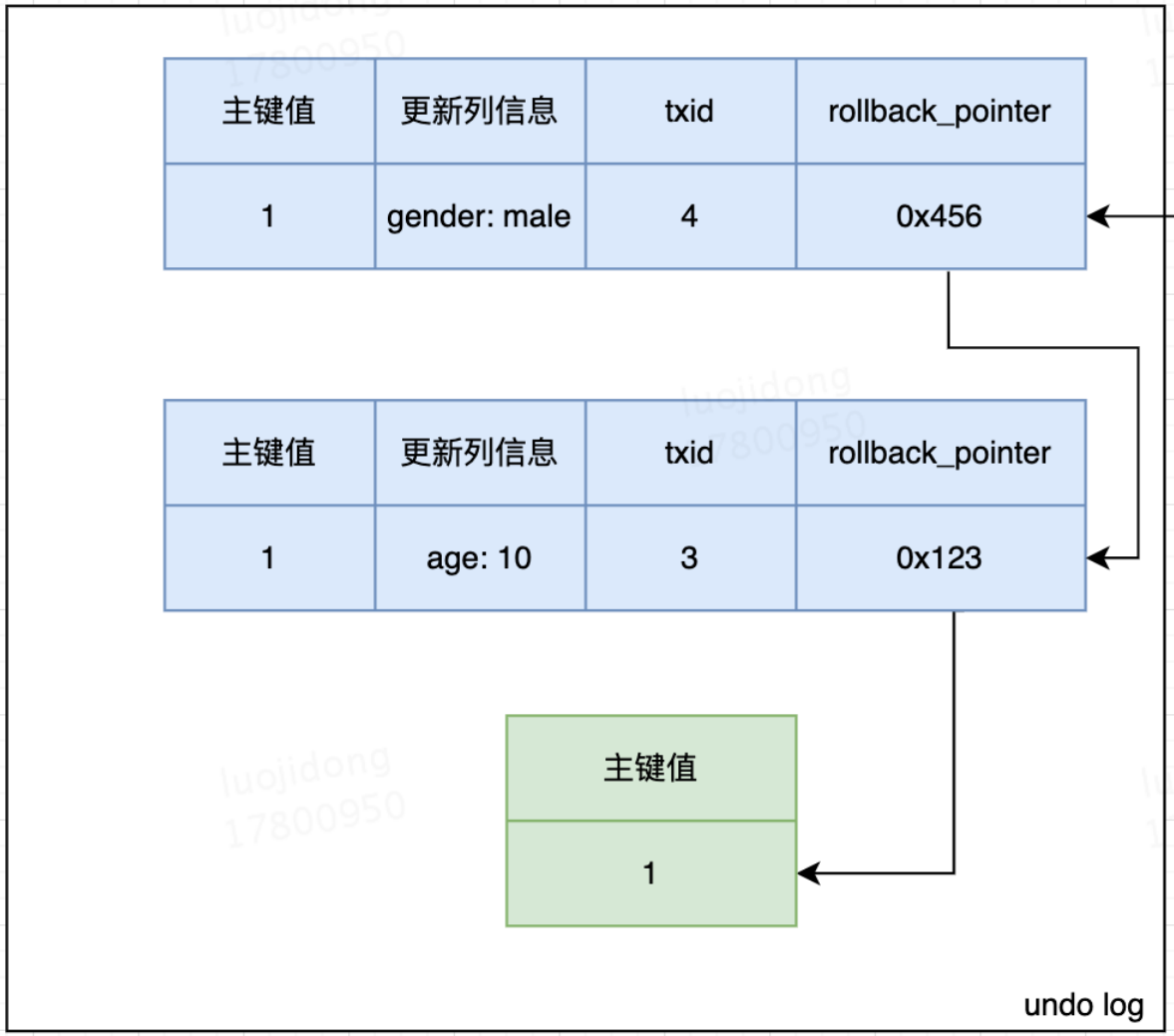
同样是在事务开始时，数据库会获取当前活跃（未提交的）的事务id列表，以及当前分配的最大事务id

同样遵循以下规则：

- 如果对数据做更改的事务id在活跃事务id列表中，那么这个更改不可见
- 如果对数据做更改的事务id大于当前分配的最大事务id，说明是后续的事务，更改不可见
- 如果对数据做更改的事务是回滚状态，更改不可见

只是我们之前使用create_by和delete_by来表示数据是由哪个事务创建，被哪个事务删除，现在变成了由txid和delete_mark来表示，当delete_mark为0时，表示数据由txid创建，当delete_mark为1时，表示数据被txid删除，同时通过rollback_pointer形成的链表来跟踪旧版本的数据，查找数据时，会在这条链表上向前追溯，直到数据的txid满足可见性规则。并且，因为我们没有在回滚日志中保留全部的信息，所以在链表上追溯时，要依次应用回滚日志中记录的修改，比如我们在更新操作中提到的，将年龄改为20，又将性别改为female

id	name	age	gender	txid	delete_mark	rollback_pointer
1	liming	20	female	5	0	0x789



这时如果事务3想读取liming这行数据，就要在最新数据上，先把gender改回male，再把age改回10，然后才是满足事务3一致性快照的数据

4.3 快照隔离和并发问题

那么快照隔离能够防止哪些并发问题呢？回顾一下我们之前提到的并发问题

- 脏写：我们之前提到了，快照隔离也是通过写加长锁来避免脏写，所以"脏写"不会出现
- 脏读：由于快照隔离的可见性规则限制了我们只能从已提交的数据中读取数据，所以"脏读"不会出现
- 不可重复读：由于快照隔离使得事务始终从一个一致性的快照中读取数据，即使数据被其他事务修改了，也不会被读取到，所以显然是可以"重复读"的，因此"不可重复读"不会出现
- 幻读：在快照隔离中，假设当前事务做了一个条件读取操作，即使其他事务的插入、更新和修改使得该条件下的数据发生了变化，由于可见性规则的作用，这些数据对当前事务也不可见，那么快照隔离是否能防止幻读？对于严格意义上的幻读，比如对于只读事务来讲，快照隔离是可以防止幻读的。但是如果根据查询结果做了写操作，例如我们上面提到的幻读导致的Write Skew，快照隔离是无法避免的，因为他并没有阻止其他事务的写操作，只是让这些写操作对当前事务不可见了
- 丢失更新：快照隔离可以避免丢失更新，我们可以针对当前事务开始后到提交前这段时间提交的这些事务，记录他们修改的数据，如果发现当前事务写的数据和这些已提交事务修改的数据有冲突，那么当前事务应该回滚，从而避免丢失更新的现象，这种方法也叫First-commiter-wins，也就是说先提交的事务会修改数据成功。但是，实际的快照隔离是否能避免丢失更新取决于数据库的实现，比如Postgresql的快照隔离是防止丢失更新的，而Mysql InnoDB的快照隔离不会阻止丢失更新
- Read Skew：和不可重复读一样，快照隔离显然可以避免Read Skew
- Write Skew：可以回顾一下我们在Write Skew中的两个例子，很明显不管是严格意义上的Write Skew，还是幻读导致的Write Skew，快照隔离都无法避免

5 Mysql隔离级别实现

下面我们来看看Mysql中的隔离级别，Mysql提供了四种隔离级别：

- 未提交读

- 提交读
- 可重复读
- 可串行化

5.1 未提交读

未提交读很简单，只是对写操作加了长锁，和我们上面说的基于锁实现未提交读隔离级别的方式是一致的，所以没啥好说的，Mysql的"未提交读"也是避免了脏写，其他问题都有可能出现

5.2 提交读

Mysql使用MVCC实现了快照隔离，这里的"提交读"隔离级别也通过MVCC进行了实现，只不过在快照隔离中，我们是一个事务一个一致性快照，而在"提交读"隔离级别下，是一条语句一个一致性快照

5.3 可重复读

Mysql的"可重复读"本质就是快照隔离，通过MVCC实现，具体的实现方式采用维护回滚日志的方式，即Mysql中的undo log

我们在前面提到了在快照隔离中，幻读和Write Skew是无法避免的，另外由于Mysql的实现，丢失更新也无法避免，如果不想切换到"可串行化"隔离级别，我们就需要手动加锁来解决这些问题，那么我们分别来看看如何避免这几个问题

既然要手动加锁，我们先了解一下Mysql中相关的锁：（下面所有的讨论都基于Mysql的"可重复读"隔离级别）

5.3.1 表锁

Mysql中的表锁包括：

- 普通的表锁
- 意向锁
- 自增锁（AUTO-INC Locks）

- MDL锁（metadata lock）

我们这里讨论前三种

5.3.1.1 普通的表锁

表锁就是对表上锁，可以对表加S锁：

```
LOCK TABLES ... READ
```

也可以对表加X锁：

```
LOCK TABLES ... WRITE
```

这俩锁的兼容性也很显而易见：

	X	S
X	互斥	互斥
S	互斥	兼容

5.3.1.2 意向锁

Mysql支持多粒度封锁，既可以锁表，也可以锁定某一行。那我们如果要加表锁，就要检查所有的数据上是否有行锁，为了避免这种开销，Mysql也引入了意向锁，要加行锁时，需要先在表上加意向锁，这样锁表时直接判断是否和意向锁冲突即可，不需要再检测所有数据上的行锁

意向锁的规则也很简单：IS和IX表示意向锁，要给行加S锁前，需要先加IS锁，要给行加X锁前，需要先加IX锁，意向锁之间不会相互阻塞

加上意向锁之后，表锁的兼容性其实也很简单：

	X	IX	S	IS
X	互斥	互斥	互斥	互斥
IX	互斥	兼容	互斥	兼容
S	互斥	互斥	兼容	兼容
IS	互斥	兼容	兼容	兼容

5.3.1.3 自增锁（AUTO-INC Locks）

自增锁很显然就是给自增id这种场景用的，也就是设置了AUTO_INCREMENT的列，插入数据时，通过加自增锁申请id，然后立即释放自增锁。自增锁跟事务关系不大，我们不再详细讨论

5.3.2 行锁

首先，Mysql的行锁并不一定是锁住某一行，也可能是锁住某个区间

Mysql中有四种行锁

- Next-Key Locks，也叫Ordinary locks，对索引项以及和上一个索引项之间的区间加锁，比如索引中有数据1，4，9，(4, 9]就是一个Next-Key Locks，Next-Key Lock是Mysql加锁的基本单位，会在一些情况下优化为下面的Record Locks或者Gap Locks
- Record Locks，也叫rec-not-gap locks，就是Next-Key Locks优化去掉了区间锁，只需要锁索引项
- Gap Locks，对两个索引项的区间加锁，比如索引中有数据1，4，9，(4, 9) 就是一个Gap Lock
- Insert intention Locks，插入意向锁，insert操作产生的Gap锁，给要插入的索引区间加锁，比如索引中有数据1，4，9，要插入5时，加插入意向锁(4, 9)

我们给出兼容性矩阵：(S锁和S锁永远是相互兼容的，下面的兼容或者互斥说的是S和X，X和S，X和X这种情形，并且锁住的行有交集）下面第一列表示已经存在的锁，第一行表示正在请求的锁

	Next-Key lock	Record lock	Gap lock	Insert intention lock
Next-Key lock	不兼容	不兼容	兼容	不兼容
Record lock	不兼容	不兼容	兼容	兼容
Gap lock	兼容	兼容	兼容	不兼容
Insert intention lock	兼容	兼容	兼容	兼容

注意这个矩阵不是完全对称的：

- Gap lock只会阻塞插入意向锁，不会和其他的锁冲突
- Next-key lock和Gap lock会阻塞插入意向锁，相反插入意向锁不会阻塞任何加锁请求

简单讨论一下各种操作会加的锁：

样例数据：表t，id为主键，c为二级索引

id	c	d
1	10	20
3	10	20
5	15	30

5.3.2.1 insert

插入时加插入意向锁，并在要插入的索引项上加Record Lock

5.3.2.2 delete/update/select ... for update

delete、update和select加X锁的情况相似，下面以delete为例说明

- 不加条件的delete/update/select ... for update：比如 delete from t 在表上加IX锁，所有的主键索引记录加Next-key lock，相当于锁表了，其他无法使用索引的条件删除都等同于这种情况
- 主键等值条件delete/update/select ... for update：比如 delete from t where id = 1 在表上加IX锁，id=1的索引记录上加Record lock
- 主键不等条件delete/update/select ... for update：比如 delete from t where id < 2 在表上加IX锁，所有访问到的索引记录（直到第一个不满足条件的值）加Next-key lock，这里就是在id=1和id=3上加Next-key lock，即锁住了(-∞, 1]和(1, 3]
- 二级索引等值条件delete/update/select ... for update：比如 delete from t where c = 10 在表上加IX锁，所有访问到的二级索引记录（直到第一个不满足条件的值）加Next-key lock，最后一个索引项优化为Gap lock，这里就是(-∞, 10]加Next-key lock，(10, 15)加Gap lock；对应的主键索引项加Record lock
- 二级索引不等条件delete/update/select ... for update：比如 delete from t where c < 10 在表上加IX锁，所有访问到的二级索引记录（直到第一个不满足条件的值）加Next-key lock，这里就是(-∞, 10]和(10, 15]加Next-key lock

5.3.2.3 select ... lock in share mode

加S锁时，覆盖索引的情况比较特殊，其他都和加X锁时相同

下面我们讨论一下覆盖索引的情况：（覆盖索引是指，查询只需要使用索引就可以查到所有数据，不必再去主键索引中查询）

假设做如下查询

```
select id from t where c = 10 lock in share mode
```

针对这个查询，Mysql的加锁规则和X锁时一样，(-∞, 10]加Next-key lock，(10, 15)加Gap lock，但是因为这个查询只需要查二级索引就可以了，Mysql不会再去主键索引查询，不查主键索引也就不会在主键索引上加锁。如果简单的把这种行锁认为是锁住了数据行，可能会出现意想不到的结果，比如在上面加S锁的情况下，跑一下下面的查询：

```
update t set d = d + 1 where id = 1
```

会发现Mysql并不会阻止你，因为这个查询根本没有用列c上的索引，又怎么会阻塞呢，但是他确实把我们之前貌似"锁住"的数据修改了

那如果想避免这种情况怎么办，可以修改查询，让索引覆盖不了；也可以把S锁换成X锁：

```
select c from t where c = 10 for update
```

加X锁的话，不管查询有没有索引覆盖，Mysql都会回去主键索引查询一下，给id=1和id=2的索引项加上锁

5.3.3 手动加锁避免并发问题

看完了锁我们再讨论一下如何通过加锁避免在"可重复读"隔离级别会出现的并发问题

我们一个用户表users作为样例数据：

id为主键，name列有非唯一索引

id	name	fans_cnt
1	liming	10
2	zhangsan	20

5.3.3.1 丢失更新

比如下面的“丢失更新”的例子：

两个事务并发给liming的粉丝数加1

时间	事务A	事务B
T1	begin	begin
T2	select fans_cnt from users where id = 1 fans_cnt = 10	select fans_cnt from users where id = 1 fans_cnt = 10
T3	update users set fans_cnt = 11 where id = 1	
T4	commit	
T5		update users set fans_cnt = 11 where id = 1
T6		commit

这里我们给读加锁就可以避免丢失更新：

时间	事务A	事务B
T1	begin	begin
T2	select fans_cnt from users where id = 1 for update fans_cnt = 10	select fans_cnt from users where id = 1 for update
T3	update users set fans_cnt = 11 where id = 1	wait
T4	commit	wait

时间	事务A	事务B
T5		fans_cnt = 11
T6		update users set fans_cnt = 12 where id = 1
		commit

事务B会等到事务A提交

当然也可以通过乐观锁的方式：

时间	事务A	事务B
T1	begin	begin
T2	select fans_cnt from users where id = 1 fans_cnt = 10	select fans_cnt from users where id = 1 fans_cnt = 10
T3	update users set fans_cnt = 11 where id = 1 and fans_cnt = 10	
T4	commit	
T5		update users set fans_cnt = 11 where id = 1 and fans_cnt = 10
T6		commit

事务B因为where条件不满足，不会更新成功，可以自己在应用代码里重试

5.3.3.2 幻读

我们之前讨论过，只读事务不会有幻读的问题，这里取幻读导致Write Skew的例子来讨论：

假设我们要求users表中name唯一，并且name上没有唯一索引

时间	事务A	事务B
T1	begin	begin
T2	select * from users where name='wangwu'	select * from users where name='wangwu'
T3	发现没有wangwu	
T4	insert into users values(wangwu, 0)	
T5	commit	
T6		发现没有wangwu
T7		insert into users values(wangwu, 0)
T8		commit

最终users表中会有两条wangwu的记录

同样，给读操作加锁就可以避免

时间	事务A	事务B
T1	begin	begin

时间	事务A	事务B
T2	select * from users where name='wangwu' lock in share mode	select * from users where name='wangwu'lock in share mode
T3	发现没有wangwu	
T4	insert into users values(wangwu, 0)	
T5	wait	
T6	wait	
T7	wait	insert into users values(wangwu, 0)
T8	success	deadlock!!!

这里事务A和事务B在读操作时会获取Gap-lock，事务A插入时请求插入意向锁被事务B的Gap lock阻塞，后面事务B插入时请求插入意向锁又被事务A的Gap lock阻塞，Mysql死锁检测机制会自动发现死锁，最终只有事务A能够插入成功

5.3.3.3 Write Skew

Write Skew 我们还是用之前信用卡账户的例子：

cards表

id为主键，列name有非唯一索引

id	name	money
1	liming	20

id	name	money
2	liming	20
3	zhangsan	10

liming有两张卡，总共40块钱，事务A和事务B分别对这两张卡扣款

时间	事务A	事务B
T1	begin	begin
T2	select * from cards where name = 'liming'	select * from cards where name = 'liming'
T3	发现还有40块钱，扣款30	
T4	update cards set money = money-30 where id = 1	
T5		发现还有40块钱，扣款30
T6		update cards set money = money-30 where id = 2
T7	commit	
T8		commit

最后发现liming只有40块钱，却花出去60

解决方法同样很简单，给读加锁就好

时间	事务A	事务B
T1	begin	begin
T2	select * from cards where name = 'liming' for update	select * from cards where name = 'liming'for update
T3	发现还有40块钱，扣款30	wait
T4	update cards set money = money-30 where id = 1	wait
T5	commit	wait
T6		发现还有10块钱，无法扣款30
T7		commit

读操作加锁之后，事务B需要阻塞到事务A提交才能完成读取，并且读到最新的数据，不会再出现liming超额花钱的情况

5.4 可串行化

Mysql的可串行化实现方式就是我们上面介绍的可串行化的加锁实现方式（即两阶段锁），可以避免上面所有的并发问题，不过两阶段锁也存在下面的问题：

- 性能差，这个很显然，加锁限制了并发，并且带来了加锁解锁的开销
- 容易死锁

另外，可串行化还有其他实现方式：

- 串行执行
- 可串行化快照隔离（Serializable Snapshot Isolation，SSI）

我们简单介绍一下

5.4.1 串行执行

> 避免代码bug最好的方式就是不写代码
>
> 避免并发问题最好的方式就是没有并发

这种实现可串行化的方式就是真的让事务串行执行，即在单线程中顺序执行，因此以这种方式实现，本身就不会有并发问题，直接实现了可串行化

这种方式有时候会比并发的方式性能更好，因为避免了加锁这种操作的开销。比如redis的事务就采用这种方式实现

这种方式也有几个明显的问题：

- 不支持交互式查询：我们在使用事务时，很多场景都是发起查询，然后根据查询结果，发起下一次查询，如果串行执行，系统执行事务的吞吐量（单位时间执行的事务数量）会受到很大影响，因为很多时间都消耗在查询结果传输这种网络IO上。因此，采用串行方式的数据库都不支持这种交互式查询的方式，如果需要在事务中实现一些业务逻辑，只能使用数据库提供的存储过程，比如在Redis中可以通过编写Lua脚本来实现
- 吞吐量受限于单核CPU：由于是单线程执行，系统的性能受限于单核CPU，不能很好地利用多核CPU
- 对IO敏感：如果数据需要从磁盘中读取，那么性能会因为磁盘IO受到很大影响

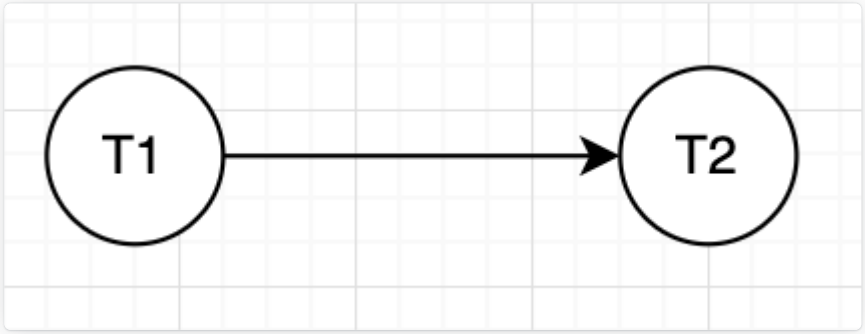
5.4.2 可串行化快照隔离

可串行化快照隔离就是在快照隔离的基础上做到了可串行化，主要思想是在快照隔离基础上增加了一种检测机制，当发现当前事务可能会导致不可串行化时，会将事务回滚。Postgresql的“可串行化”隔离级别采用了这种实现方式

我们简单介绍一下这种检测机制：

事务之间的关系我们可以用图结构来表示，图中的顶点是事务，边是事务之间的依赖关系，这就是多版本可序列化图（Multi-Version Serialization Graph，MVSG）

其中边是有向边，通过事务对相同数据的读写操作来定义，比如T1将x修改为1之后，T2将x修改为2，T1和T2之间的关系可以这样表示：



边的方向由T1指向T2表示T1在T2之前发生

事务之间可能发生冲突的依赖关系有三种，也就是图中边的种类有三种：

- 写写依赖（ww-dependencies）：T1为数据写入新版本，T2用更新的版本替换了T1，则T1和T2构成写写依赖：$T_1 \xrightarrow{ww} T_2$
- 写读依赖（wr-dependencies）：T1为数据写入新版本，T2读取了这个版本的数据，或者通过谓词读（通过条件查询）的方式读取到这个数据，则T1和T2构成写读依赖：$T_1 \xrightarrow{wr} T_2$
- 读写反依赖（rw-dependencies）：T2为数据写入新版本，而T1读取了旧版本的数据，或者通过谓词读（通过条件查询）的方式读取到这个数据，则T1和T2构成读写反依赖：$T_1 \xrightarrow{rw} T_2$
- 因为读读并发没有任何问题，所以我们这里没有读读依赖

由事务为顶点，上面三种依赖关系为边，可以构成一个有向图，如果这个有向图存在环，则事务不可串行化，因此可以通过检测环的方式，来回滚相关事务，做到可串行化。但是这样做开销比较大，因此学术界提了一条定理：如果存在环，则图中必然存在这样的结构：

$T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$，因此可以通过检测这种"危险结构"来实现，当然，这样实现的话可能会错误的回滚一些事务，这里关于定理的证明以及危险结构的检测算法我们就不再介绍了，感兴趣的话可以看看相关资料

6 参考文献

【1】Designing Data-Intensive Applications

【2】A Critique of ANSI SQL Isolation Levels

【3】Mysql Reference Manual

【4】数据库事务处理的艺术