1.事务隔离性的基本概念

1.1 什么是ACID中的Isolation,隔离性

Isolation,隔离性，也有人称之为并发控制（concurrency control）。事务的隔离性要求每个事务读写的对象对其他事务都是相互隔离的，也就是这个事务提交前，这个事务的修改内容对其他事务都是不可见的。事务的隔离性，主要是解决不同事物之间的相互读写影响。

所谓的读写影响注意分为三种：

- 脏读：读到了别的事务尚未提交（commit）的变更，别人没提交，我读到了。
- 不可重复读：别的事务提交了变更，被当前事务读到了。然后导致本事务多次select的结果不一样，读到了别的事务提交的内容。
- 幻读：也是读到了别的事务提交的内容，但是跟上面的不同之处在于，读到了原本不存在的记录。

注意，不可重复读，主要是读到了别的事务update的内容。而幻读，是读到了别的事务insert的内容。

1.2 隔离性的隔离级别

为了解决事务隔离性的问题，数据库一般会有不同的隔离级别来解决相应的读写影响。

- 读未提交：一个事务B还没提交，它的修改就被别的事务A读到了。
- 读已提交：一个事务B提交后，它的修改被其他事务A看到了。
- 可重复读：一个事物B提交前和提交后，事务A都无法读到事务B的变更。
- 串行化：对同一行记录，当出现不同事物的读写冲突时，是通过串行化的方式解决的，后一个事务必须等前一个事务完成才能执行。

不同隔离级别能够解决不同的隔离性问题。

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

需要注意的是，这是标准事务隔离级别的定义。在MySQL的innodb引擎中，在可重复读级别下，通过mvcc解决了幻读的问题，具体实现我们后面再讲。

同时，需要注意的是，到目前为止，我们说的读，都是”快照读”，普通的select。后面我们还会提到“当前读”，是不一样的哦。

2.事务隔离性的实现

要实现事务的隔离性，需要了解两个方面的内容，一个是锁，一个是多版本并发控制（MVCC）。

2.1 事务的行锁

InnoDB中，实现了两种标准的行级锁：

- 共享锁（S Lock），也叫读锁，允许事务读取一行数据。
- 排它锁（X Lock），也叫写锁，允许事务删除或者更新一行数据（注意，这里没有提到插入哦，插入涉及到幻读，可以看文章最后的说明）

普通select语句不会有任何锁，那么如何获得共享锁和排它锁呢?

- Select … lock in share mode语句能够获得共享锁
- Select … for update（特殊的select，用mysql简单实现分布式锁经常用它）、Update、delete语句能够获得排它锁

当一个事务A已经获得了行r的共享锁，那么另一个事务B可以立刻获得行r的共享锁，因为不会改变r的数值，这种叫做锁兼容。

如果这时候有事务C希望获得行r的排它锁，那么就必须等待事务A和事务B释放行r的共享锁之后，才能获得排它锁，这种叫做锁不兼容。

	共享锁（读锁）	排它锁
共享锁（读锁）	不冲突	互斥
排它锁	互斥	互斥

普通的select不会对行上锁，而select…lock in share mode会上共享锁，select…for update会上排它锁。

- 对于普通的select的读取方式，称为”快照读“，也叫”一致性非锁定读“。
- 对于带锁的select读取，或者update tb set a = a+1（读取a的当前值），称为“当前读”，也叫“一致性锁定读”。

如果在update、insert的时候，不能进行select，那么服务的并发访问性能就太差了。因此，我们日常的查询，都是“快照读”，不会上锁，只有在update\insert\“当前读”的时候，才会上锁。而为了解决“快照读”的并发访问问题，就引入了MVCC。

2.2 多版本并发控制MVCC

如果说上面的行锁是一种悲观锁，那么MVCC就是一种乐观锁的实现方式，而且是一种很常用的乐观锁实现方式。

所谓多版本，就是一行记录在数据库中存储了多个版本，每个版本以事务ID作为版本号。InnoDB 里面每个事务有一个唯一的事务 ID，是在事务开始的时候向InnoDB的事务系统申请的，并且按照申请顺序严格递增的。假如一行记录被多个事务更新，那么，就会产生多个版本的记录。

以某一行数据作为例子：

SQL	Tx_id	Value
	15	22
Set value = 17 (tx_id = 25)		
	25	17
Set value = value + 2 (tx_id = 30)		
	30	19 阿九笔记

经过两次事务的操作，value从22变成了19，同时，保留了三个事务id，15、25、30。

在每个记录多版本的基础上，需要利用“一致性视图”，来做版本的可见性判断。

这里，我们要区分MySQL里面的两个”视图”概念：

- 一个是view，通过语法create view … 实现，主要创建一个虚拟表，用来执行查询语句。
- 一个是InnoDB用来实现mvcc的一致性视图（consistent read view），纯逻辑概念，没有物理结构，定义了在事务期间，你能看到哪些版本的数据。

我们全文提到的“视图”都是第二种，主要是支持InnoDB在“读已提交”和“可重复读”级别的并发访问问题。

- “读未提及”级别下，没有一致性视图
- “读已提交”级别下，会在 **每个SQL开始执行的时候** 创建一致性视图
- “可重复读”级别下，会在 **每个事务开始的时候** 创建一致性视图
- “串行化”级别下，直接通过加锁避免并发问题

下面，我们简单介绍一下创建一致性视图的逻辑。

以“可重复读”级别为例。

1. 当一个事务开启的时候，会向系统申请一个新事务id
2. 此时，可能还有多个正在进行的其他事务没有提交，因此在瞬时时刻，是有多个活跃的未提交事务id
3. 将这些未提交的事务id组成一个数组，数组里面最小的事务id记录为低水位，当前系统创建过的事务id的最大值+1记录为高水位
4. 这个数组array 和 高水位，就组成了“一致性视图”。

有了一致性视图后，我们就可以判断一行数据的多版本可见性了，无论是“读已提交”还是“可重复读”级别，可见性判断规则是一样的，区别在于创建快照（一致性视图）的时间。

在当前事务中，读取其他某一行的记录，对其中的版本号的可见性判断有五种情况（建议自己跟着捋一捋，挺重要的）：

1. 如果版本号小于“低水位”，说明事务已经提交，那肯定 可见；
2. 如果版本号大于“高水位”，说明这行数据的这个事务id版本是在快照后产生的，那肯定 不可见；
3. 如果版本号在事务数组array中，说明这个事务还没提交，所以 不可见；
4. 如果版本号不在事务数组array中，且低于高水位，说明这个事务已经提交，所以 可见；
5. 当然，无论什么时候，自己的事务id中的任何变化，都是可见的

可以看看下面这个例子，更容易理解。

系统创建过的事务id：1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

事务A启动，拍个快照

此时未提交的事务id有：7，8，9

一致性视图：数组array[7,8,9] + 高水位16（15+1）

对于任意一行数据的可见性判断：

- 小于7的，可见
- 大于16的，说明是快照后产生的，不可见
- 10-15，不在数组array中，说明已经提交了，可见
- 7，8，9在array中，说明未提交，不可见

两个重要结论：

- **InnoDB 利用了“所有数据都有多个版本”的这个特性，实现了“秒级创建快照”的能力。**
- **MVCC的实现，就是根据当前事务的事务id为依据创建“一致性视图”，利用一致性视图来判断数据版本的可见性。**

3.隔离性实战

下面，我们来两个实战案例，将上面的基础概念与实现融会贯通吧。

1）并发select&update 案例

id=1 的value初始为1。

	事务A	事务B
Time1	Begin;	
Time2	Select value from tb where id = 1;	Begin;
Time3		Select value from tb where id = 1;
Time4		Update tb set value = 2 where id = 1;
Time5	Select value from tb where id = 1;	
Time6		Commit;
Time7	Select value from tb where id = 1;	
Time8	Commit;	
Time9	Select value from tb where id = 1;	

我们看下，在不同隔离级别，Time5、Time7、Time9事务A查询到的value 分布为多少。

- “读未提交”：2，2，2
- “读以提交”：1，2，2
- “可重复读”：1，1，2
- 串行化：1，1，2（注意，这里在事务A提交前，事务B都会阻塞，直到事务A提交后才能执行）
-

2）并发update案例

id=1 的value初始为1，在可重复读级别：

事务A	事务B	事务C
start transaction with consistent snapshot;		
	start transaction with consistent snapshot;	
		Update tb set value = value + 1 where id = 1;
	Update tb set value = value + 1 where id = 1;	
	Select value from tb where id = 1;	
Select value from tb where id = 1;		
Commit;		
	Commit;	

我们看一下，你猜猜事务A和事务B读取的value是多少?

答案是：1 和 3

可能会产生困惑，事务A在启动后快照，所以读到了1是正常的，但是事务2在启动的时候快照了，然后在自己的事务中+1，怎么会读到3而不是2呢?

原因很简单，即使是在可重复读的级别，事务 **更新数据** 的时候，只能用**当前读**（想想也能理解，不然update就出现数据不一致了）。

如果当前的记录的行锁被其他事务占用的话，就需要进入锁等待。而读提交的逻辑和可重复读的逻辑类似，它们最主要的区别是：在可重复读隔离级别下，只需要在事务开始的时候创建一致性视图，之后事务里的其他查询都共用这个一致性视图；在读提交隔离级别下，每一个语句执行前都会重新算出一个新的视图。

这里，我们需要注意的是事务的启动时机。

- begin/start transaction 命令并不是一个事务的起点，在执行到它们之后的第一个操作 InnoDB 表的语句，事务才真正启动,一致性视图是在执行第一个快照读语句时创建的。

- 如果你想要马上启动一个事务，可以使用 start transaction with consistent snapshot 这个命令，一致性视图是在执行 start transaction with consistent snapshot 时创建的。

4.关于幻读

前文已经提到了，对于普通数据库，需要到可串行化的隔离级别才能解决幻读问题。

而对于InnoDB存储引擎来说，在可重复读级别下就能解决幻读问题。

InnoDB存储引擎有三种行锁算法：

- 行锁：当个行记录上的锁
- 间隙锁：Gap Lock，锁定一个范围，但不包含记录本身
- Next-Key Lock:就是行锁+间隙锁，同时锁上一个范围，并且锁定记录本身