补充：前缀索引

当字段值比较长的时候，建立索引会消耗很多的空间，搜索起来也会很慢。我们可以通过截取字段的前面一部分内容建立索引，这个就叫前缀索引。

创建一张商户表，因为地址字段比较长，在地址字段上建立前缀索引：

```
create table shop(address varchar(120) not null);
alter table shop add key (address(12));
```

问题是，截取多少呢？截取得多了，达不到节省索引存储空间的目的，截取得少了，重复内容太多，字段的散列度（选择性）会降低。怎么计算不同的长度的选择性呢？

先看一下字段在全部数据中的选择度：

```
select count(distinct address) / count(*) from shop;
```

通过不同长度去计算，与全表的选择性对比：

```
select count(distinct left(address,10))/count(*) as sub10,
count(distinct left(address,11))/count(*) as sub11,
count(distinct left(address,12))/count(*) as sub12,
count(distinct left(address,13))/count(*) as sub13
from shop;
```

只要截取前 13 个字段，就已经有比较高的选择性了（这里的数据只是举例）。

课程目标

1、 掌握事务的特性与事务并发造成的问题

2、 事务读一致性问题的解决方案

3、 MVCC 的原理

4、 锁的分类、行锁的原理、行锁的算法

内容定位

适合了解 MySQL 和 InnoDB 架构、清楚索引本质的同学学习。

课程集中答疑链接：https://gper.club/articles/7e7e7f7ff7g55gc8g6c

1 什么是数据库的事务？

1.1 事务的典型场景

在项目里面，什么地方会开启事务，或者配置了事务？无论是在方法上加注解，还是配置切面。

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
<tx:attributes>
<tx:method name="save*" rollback-for="Throwable" />
<tx:method name="add*" rollback-for="Throwable" />
<tx:method name="send*" rollback-for="Throwable" />
<tx:method name="insert*" rollback-for="Throwable" />
</tx:attributes>
</tx:advice>
```

比如下单，会操作订单表，资金表，物流表等等，这个时候我们需要让这些操作都在一个事务里面完成。当一个业务流程涉及多个表的操作的时候，我们希望它们要么是全部成功的，要么都不成功，这个时候我们会启用事务。

在金融的系统里面事务配置是很常见的，比如行内转账的这种操作，如果我们把它

简单地理解为一个账户的余额增加，另一个账户的余额减少的情况（当然实际上要比这复杂），那么这两个动作一定是同时成功或者同时失败的，否则就会造成银行的会计科目不平衡。

1.2 事务的定义

什么是事务？

维基百科的定义：事务是数据库管理系统（DBMS）执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。

这里面有两个关键点，第一个，它是数据库最小的工作单元，是不可以再分的。第二个，它可能包含了一个或者一系列的 DML 语句，包括 insert delete update。

（单条 DDL（create drop）和 DCL（grant revoke）也会有事务）

1.3 哪些存储引擎支持事务

在我们第一天的课里面说到了，InnoDB 支持事务，这个也是它成为默认的存储引擎的一个重要原因：

https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html

另一个是 NDB。

1.4 事务的四大特性

事务的四大特性：ACID。

第一个，原子性，Atomicity，也就是我们刚才说的不可再分，也就意味着我们对数据库的一系列的操作，要么都是成功，要么都是失败，不可能出现部分成功或者部分失败的情况。以转账的场景为例，一个账户的余额减少，对应一个账户的增加，这两个一定是同时成功或者同时失败的。

全部成功比较简单，问题是如果前面一个操作已经成功了，后面的操作失败了，怎么让它全部失败呢？这个时候我们必须要回滚。

原子性 ,在 InnoDB 里面是通过 undo log 来实现的 ,它记录了数据修改之前的值(逻辑日志），一旦发生异常，就可以用 undo log 来实现回滚操作。

第二个，一致性，consistent，指的是数据库的完整性约束没有被破坏，事务执行的前后都是合法的数据状态。比如主键必须是唯一的，字段长度符合要求。

除了数据库自身的完整性约束，还有一个是用户自定义的完整性。

比如说转账的这个场景，A 账户余额减少 1000，B 账户余额只增加了 500，这个时候因为两个操作都成功了，按照我们对原子性的定义，它是满足原子性的， 但是它没有满足一致性，因为它导致了会计科目的不平衡。

还有一种情况，A 账户余额为 0，如果这个时候转账成功了，A 账户的余额会变成 -1000，虽然它满足了原子性的，但是我们知道，借记卡的余额是不能够小于 0 的，所以也违反了一致性。用户自定义的完整性通常要在代码中控制。

第三个，隔离性，Isolation，我们有了事务的定义以后，在数据库里面会有很多的事务同时去操作我们的同一张表或者同一行数据，必然会产生一些并发或者干扰的操作，那么我们对隔离性的定义，就是这些很多个的事务，对表或者行的并发操作，应该是透明的，互相不干扰的。通过这种方式，我们最终也是保证业务数据的一致性。

最后一个叫做持久性，Durable，事务的持久性是什么意思呢？我们对数据库的任意的操作，增删改，只要事务提交成功，那么结果就是永久性的，不可能因为我们系统宕机或者重启了数据库的服务器，它又恢复到原来的状态了。这个就是事务的持久性。

持久性怎么实现呢？数据库崩溃恢复（crash-safe）是通过什么实现的？

持久性是通过 redo log 和 double write 双写缓冲来实现的，我们操作数据的时候，

会先写到内存的 buffer pool 里面，同时记录 redo log，如果在刷盘之前出现异常，在重启后就可以读取 redo log 的内容，写入到磁盘，保证数据的持久性。

当然，恢复成功的前提是数据页本身没有被破坏，是完整的，这个通过双写缓冲（double write）保证。

原子性，隔离性，持久性，最后都是为了实现一致性。

1.5 数据库什么时候会出现事务

无论是我们在 Navicat 的这种工具里面去操作，还是在我们的 Java 代码里面通过 API 去操作，还是加上@Transactional 的注解或者 AOP 配置，其实最终都是发送一个指令到数据库去执行，Java 的 JDBC 只不过是把这些命令封装起来了。

我们先来看一下我们的操作环境。版本（5.7），存储引擎（InnnoDB），事务隔离级别（RR）。

```
select version();

show variables like '%engine%';

show global variables like "tx_isolation";
```

执行这样一条更新语句的时候，它有事务吗？

```
update student set sname = '猫老公 111' where id=1;
```

实际上，它自动开启了一个事务，并且提交了，所以最终写入了磁盘。

这个是开启事务的第一种方式，自动开启和自动提交。

InnoDB 里面有一个 autocommit 的参数（分成两个级别， session 级别和 global 级别）。

```
show   variables like 'autocommit';
```

它的默认值是 ON。autocommit 这个参数是什么意思呢？是否自动提交。如果它的值是 true/on 的话，我们在操作数据的时候，会自动开启一个事务，和自动提交事务。

否则，如果我们把 autocommit 设置成 false/off，那么数据库的事务就需要我们手动地去开启和手动地去结束。

手动开启事务也有几种方式，一种是用 begin；一种是用 start transaction。

那么怎么结束一个事务呢？我们结束也有两种方式，第一种就是提交一个事务，commit；还有一种就是 rollback，回滚的时候，事务也会结束。还有一种情况，客户端的连接断开的时候，事务也会结束。

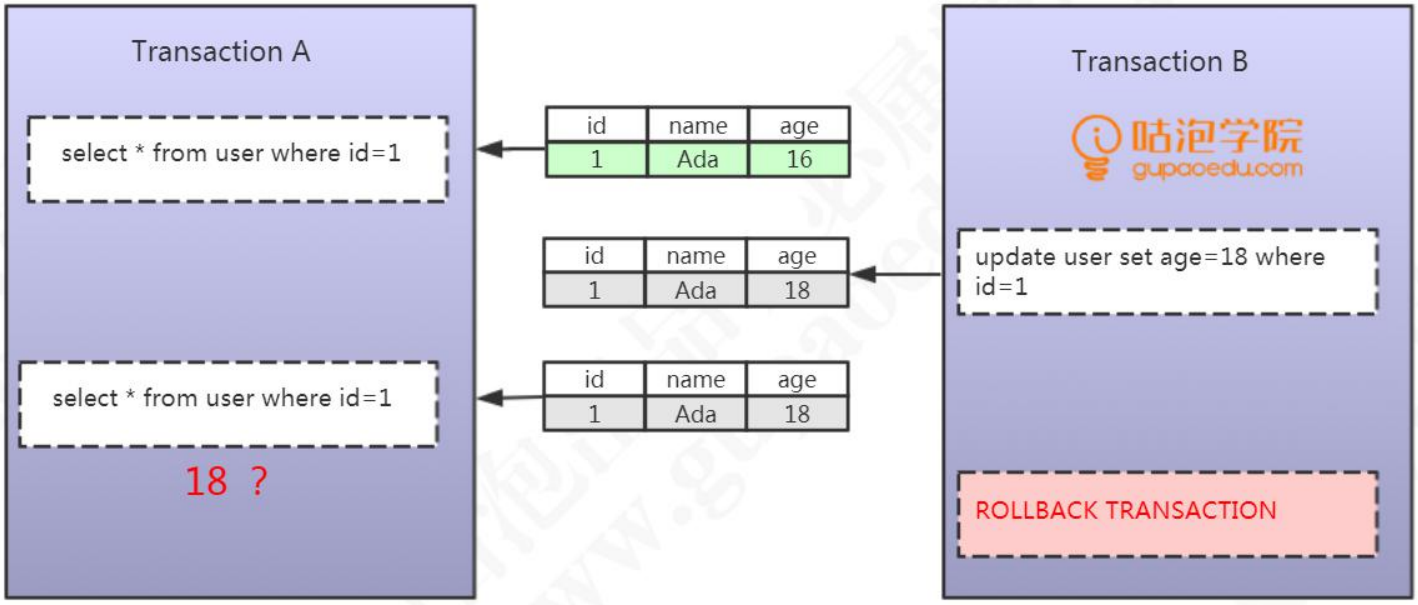
后面我们会讲到，当我们结束一个事务的时候，事务持有的锁就会被释放，无论是提交还是回滚。

我们用 begin 手工开启一个事务，执行第二个 update，但是数据没有写入磁盘，因为事务还没有提交，这个时候 commit 一下，再刷新一下，OK，写入了。

这个就是我们开启和结束事务的两种方式。

1.6 事务并发会带来什么问题？

当很多事务并发地去操作数据库的表或者行的时候，如果没有我们刚才讲的事务的 Isolation 隔离性的时候，会带来哪些问题呢？



我们有两个事务，一个是 Transaction A，一个是 Transaction B，在第一个事务里面，它首先通过一个 where id=1 的条件查询一条数据，返回 name=Ada，age=16 的这条数据。然后第二个事务，它同样地是去操作 id=1 的这行数据，它通过一个 update 的语句，把这行 id=1 的数据的 age 改成了 18，但是注意，它没有提交。

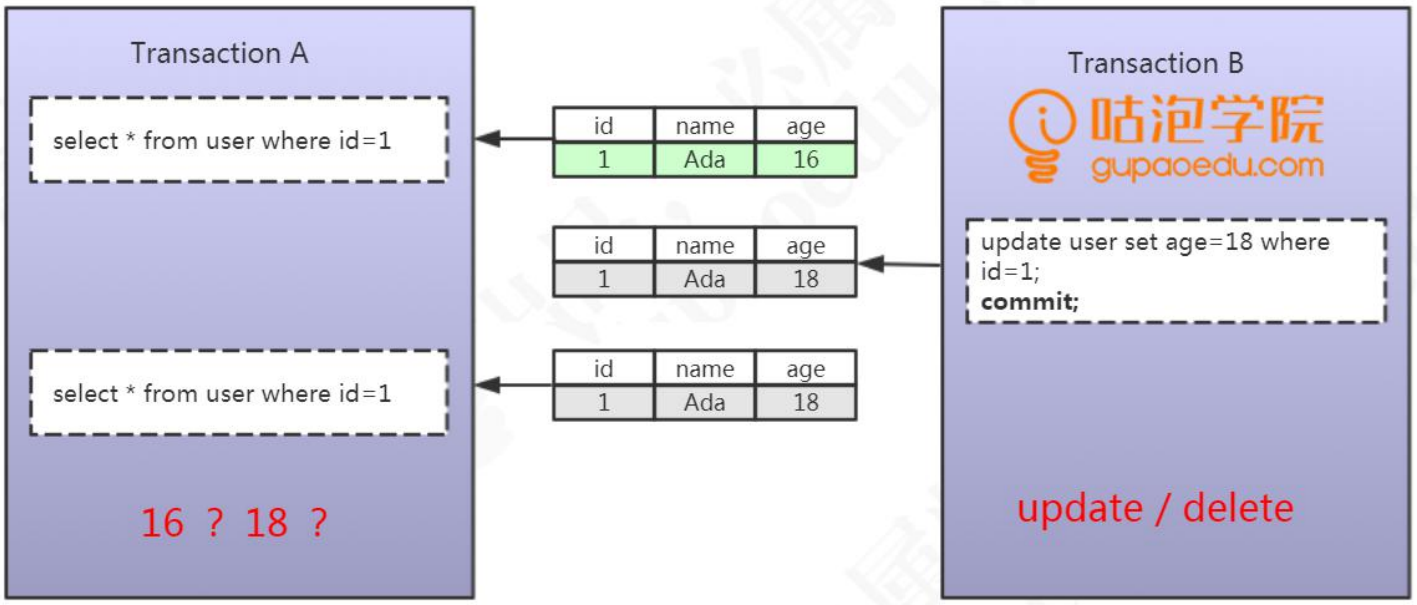
这个时候，在第一个事务里面，它再次去执行相同的查询操作，发现数据发生了变化，获取到的数据 age 变成了 18。那么，这种在一个事务里面，由于其他的时候修改了数据并且没有提交，而导致了前后两次读取数据不一致的情况，这种事务并发的问题，我们把它定义成什么？

这个叫做脏读。

如果在转账的案例里面，我们第一个事务基于读取到的第二个事务未提交的余额进行了操作，但是第二个事务进行了回滚，这个时候就会导致数据不一致。

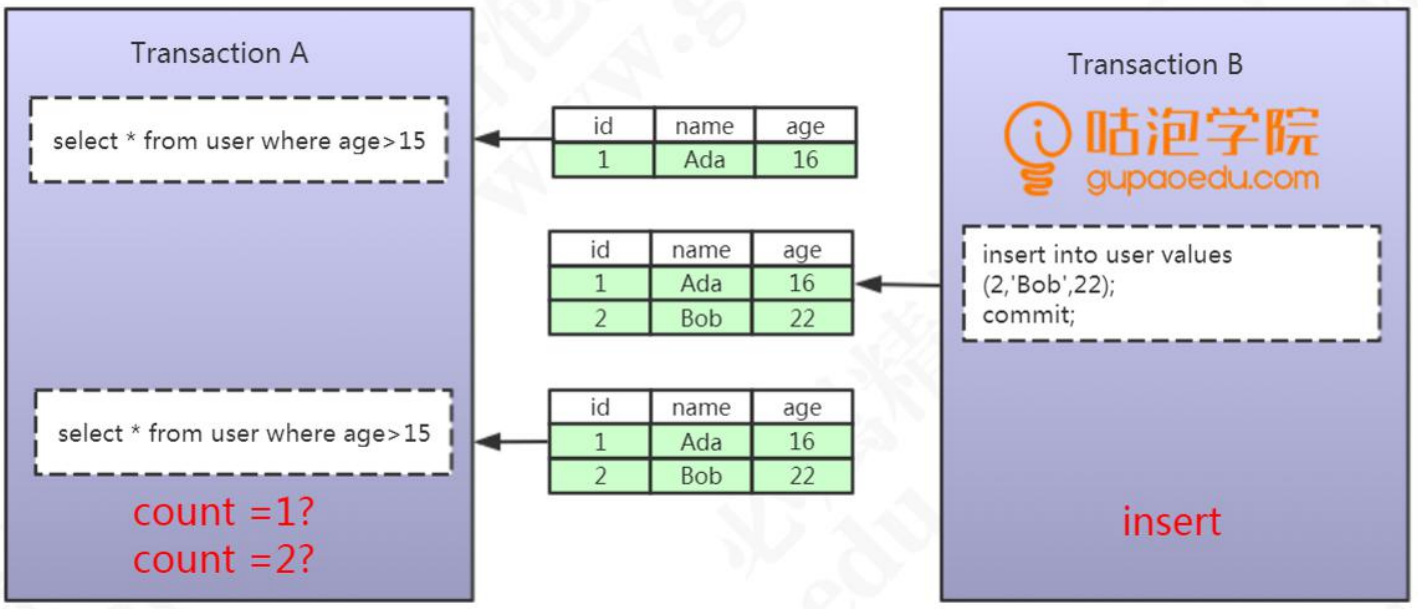
这种读取到其他事务未提交的数据的情况，我们把它叫做脏读。

我们再来看第二个。



同样是两个事务，第一个事务通过 id=1 查询到了一条数据。然后在第二个事务里面执行了一个 update 操作，这里大家注意一下，执行了 update 以后它通过一个 commit 提交了修改。然后第一个事务读取到了其他事务已提交的数据导致前后两次读取数据不一致的情况，就像这里，age 到底是等于 16 还是 18，那么这种事务并发带来的问题，我们把它叫做什么？

这种一个事务读取到了其他事务已提交的数据导致前后两次读取数据不一致的情况，我们把它叫做不可重复读。



在第一个事务里面我们执行了一个范围查询，这个时候满足条件的数据只有一条。在第二个事务里面，它插入了一行数据，并且提交了。重点：插入了一行数据。在第一个事务里面再去查询的时候，它发现多了一行数据。这种情况，我们把它叫做什么呢？

一个事务前后两次读取数据数据不一致，是由于其他事务插入数据造成的，这种情况我们把它叫做幻读。

不可重复读和幻读，的区别在那里呢？

不可重复读是修改或者删除，幻读是插入。

小结：我们刚才讲了事务并发带来的三大问题，现在来给大家总结一下。无论是脏读，还是不可重复读，还是幻读，它们都是数据库的读一致性的问题，都是在一个事务里面前后两次读取出现了不一致的情况。

读一致性的问题，必须要由数据库提供一定的事务隔离机制来解决。就像我们去饭店吃饭，基本的设施和卫生保证都是饭店提供的。那么我们使用数据库，隔离性的问题也必须由数据库帮助我们来解决。

1.7 SQL92 标准

所以，就有很多的数据库专家联合制定了一个标准，也就是说建议数据库厂商都按照这个标准，提供一定的事务隔离级别，来解决事务并发的问题，这个就是 SQL92 标准。

我们来看一下 SQL92 标准的官网。

http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

这里面有一张表格（搜索_iso），里面定义了四个隔离级别，右边的 P1 P2 P3 就是代表事务并发的 3 个问题，脏读，不可重复读，幻读。Possible 代表在这个隔离级别下，这个问题有可能发生，换句话说，没有解决这个问题。Not Possible 就是解决了这个问

题。

我们详细地分析一下这 4 个隔离级别是怎么定义的。

第一个隔离级别叫做：Read Uncommitted（未提交读），一个事务可以读取到其他事务未提交的数据，会出现脏读，所以叫做 RU，它没有解决任何的问题。

第二个隔离级别叫做：Read Committed（已提交读），也就是一个事务只能读取到其他事务已提交的数据，不能读取到其他事务未提交的数据，它解决了脏读的问题，但是会出现不可重复读的问题。

第三个隔离级别叫做：Repeatable Read (可重复读)，它解决了不可重复读的问题，也就是在同一个事务里面多次读取同样的数据结果是一样的，但是在这个级别下，没有定义解决幻读的问题。

最后一个就是：Serializable（串行化），在这个隔离级别里面，所有的事务都是串行执行的，也就是对数据的操作需要排队，已经不存在事务的并发操作了，所以它解决了所有的问题。

这个是 SQL92 的标准，但是不同的数据库厂商或者存储引擎的实现有一定的差异，比如 Oracle 里面就只有两种 RC（已提交读）和 Serializable（串行化）。那么 InnoDB 的实现又是怎么样的呢？

1.8 MySQL InnoDB 对隔离级别的支持

在 MySQL InnoDB 里面，不需要使用串行化的隔离级别去解决所有问题。那我们来看一下 MySQL InnoDB 里面对数据库事务隔离级别的支持程度是什么样的。

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对 InnoDB 不可能
串行化（Serializable）	不可能	不可能	不可能

InnoDB 支持的四个隔离级别和 SQL92 定义的基本一致，隔离级别越高，事务的并发度就越低。唯一的区别就在于，InnoDB 在 RR 的级别就解决了幻读的问题。这个也是 InnoDB 默认使用 RR 作为事务隔离级别的原因，既保证了数据的一致性，又支持较高的并发度。

1.9 两大实现方案

那么大家想一下，如果要解决读一致性的问题，保证一个事务中前后两次读取数据结果一致，实现事务隔离，应该怎么做？我们有哪一些方法呢？你的思路是什么样的呢？

总体上来说，我们有两大类的方案。

1.9.1 LBCC

第一种，我既然要保证前后两次读取数据一致，那么我读取数据的时候，锁定我要操作的数据，不允许其他的事务修改就行了。这种方案我们叫做基于锁的并发控制 Lock Based Concurrency Control（LBCC）。

如果仅仅是基于锁来实现事务隔离，一个事务读取的时候不允许其他时候修改，那就意味着不支持并发的读写操作，而我们的大多数应用都是读多写少的，这样会极大地影响操作数据的效率。

1.9.2 MVCC

所以我们还有另一种解决方案，如果要让一个事务前后两次读取的数据保持一致，那么我们可以在修改数据的时候给它建立一个备份或者叫快照，后面再来读取这个快照就行了。这种方案我们叫做多版本的并发控制 Multi Version Concurrency Control（MVCC）。

MVCC 的核心思想是： 我可以查到在我这个事务开始之前已经存在的数据，即使它在后面被修改或者删除了。在我这个事务之后新增的数据，我是查不到的。

问题：这个快照什么时候创建？读取数据的时候，怎么保证能读取到这个快照而不是最新的数据？这个怎么实现呢？

InnoDB 为每行记录都实现了两个隐藏字段：

DB_TRX_ID，6 字节：插入或更新行的最后一个事务的事务 ID，事务编号是自动递增的（我们把它理解为创建版本号，在数据新增或者修改为新数据的时候，记录当前事务 ID）。

DB_ROLL_PTR，7 字节：回滚指针（我们把它理解为删除版本号，数据被删除或记录为旧数据的时候，记录当前事务 ID）。

我们把这两个事务 ID 理解为版本号。

https://www.processon.com/view/link/5d29999ee4b07917e2e09298 MVCC 演示图

第一个事务，初始化数据（检查初始数据）

Transaction 1
begin; insert into mvcctest values(NULL,'qingshan') ; insert into mvcctest values(NULL,'jack') ; commit;

此时的数据，创建版本是当前事务 ID，删除版本为空：

id	name	创建版本	删除版本
1	qingshan	1	undefined
2	jack	1	undefined

第二个事务，执行第 1 次查询，读取到两条原始数据，这个时候事务 ID 是 2：

Transaction 2
begin; select * from mvcctest ; -- (1) 第一次查询

第三个事务，插入数据：

Transaction 3
begin; insert into mvcctest values(NULL,'tom') ; commit;

此时的数据，多了一条 tom，它的创建版本号是当前事务编号，3：

id	name	创建版本	删除版本
1	qingshan	1	undefined
2	jack	1	undefined
3	tom	3	undefined

第二个事务，执行第 2 次查询：

Transaction 2
select * from mvcctest ; (2) 第二次查询

MVCC 的查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

也就是不能查到在我的事务开始之后插入的数据，tom 的创建 ID 大于 2，所以还是只能查到两条数据。

第四个事务，删除数据，删除了 id=2 jack 这条记录：

Transaction 4
begin;

```
delete from mvcctest where id=2;
commit;
```

此时的数据，jack 的删除版本被记录为当前事务 ID，4，其他数据不变：

id	name	创建版本	删除版本
1	qingshan	1	undefined
2	jack	1	4
3	tom	3	undefined

在第二个事务中，执行第 3 次查询：

Transaction 2
select * from mvcctest ; (3) 第三次查询

查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

也就是，在我事务开始之后删除的数据，所以 jack 依然可以查出来。所以还是这两条数据。

第五个事务，执行更新操作，这个事务事务 ID 是 5：

Transaction 4
begin; update mvcctest set name ='盆鱼宴' where id=1; commit;

此时的数据，更新数据的时候，旧数据的删除版本被记录为当前事务 ID 5（undo），产生了一条新数据，创建 ID 为当前事务 ID 5：

id	name	创建版本	删除版本
1	qingshan	1	5
2	jack	1	4
3	tom	3	undefined
1	盆鱼宴	5	undefined

第二个事务，执行第 4 次查询：

Transaction 2

```
select * from mvcctest ;  (4) 第四次查询
```

查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

因为更新后的数据 penyuyan 创建版本大于 2，代表是在事务之后增加的，查不出来。

而旧数据 qingshan 的删除版本大于 2，代表是在事务之后删除的，可以查出来。

通过以上演示我们能看到，通过版本号的控制，无论其他事务是插入、修改、删除，第一个事务查询到的数据都没有变化。

在 InnoDB 中，MVCC 是通过 Undo log 实现的。

Oracle、Postgres 等等其他数据库都有 MVCC 的实现。

需要注意，在 InnoDB 中，MVCC 和锁是协同使用的，这两种方案并不是互斥的。

第一大类解决方案是锁，锁又是怎么实现读一致性的呢？

2 MySQL InnoDB 锁的基本类型

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html

官网把锁分成了 8 类。所以我们把前面的两个行级别的锁（Shared and Exclusive Locks），和两个表级别的锁（Intention Locks）称为锁的基本模式。

后面三个 Record Locks、Gap Locks、Next-Key Locks，我们把它们叫做锁的算法，也就是分别在什么情况下锁定什么范围。

2.1 锁的粒度

我们讲到 InnoDB 里面既有行级别的锁，又有表级别的锁，我们先来分析一下这两

种锁定粒度的一些差异。

表锁，顾名思义，是锁住一张表；行锁就是锁住表里面的一行数据。锁定粒度，表锁肯定是大于行锁的。

那么加锁效率，表锁应该是大于行锁还是小于行锁呢？大于。为什么？表锁只需要直接锁住这张表就行了，而行锁，还需要在表里面去检索这一行数据，所以表锁的加锁效率更高。

第二个冲突的概率？表锁的冲突概率比行锁大，还是小？

大于，因为当我们锁住一张表的时候，其他任何一个事务都不能操作这张表。但是我们锁住了表里面的一行数据的时候，其他的事务还可以来操作表里面的其他没有被锁定的行，所以表锁的冲突概率更大。

表锁的冲突概率更大，所以并发性能更低，这里并发性能就是小于。

nnoDB 里面我们知道它既支持表锁又支持行锁 ,另一个常用的存储引擎 MyISAM 支持什么粒度的锁？这是第一个问题。第二个就是 InnoDB 已经支持行锁了，那么它也可以通过把表里面的每一行都锁住来实现表锁，为什么还要提供表锁呢？

要搞清楚这个问题，我们就要来了解一下 InnoDB 里面的基本的锁的模式（lock mode），这里面有两个行锁和两个表锁。

2.2 共享锁

第一个行级别的锁就是我们在官网看到的 Shared Locks （共享锁），我们获取了一行数据的读锁以后，可以用来读取数据，所以它也叫做读锁，注意不要在加上了读锁以后去写数据，不然的话可能会出现死锁的情况。而且多个事务可以共享一把读锁。那

怎么给一行数据加上读锁呢？

我们可以用 select …… lock in share mode; 的方式手工加上一把读锁。

释放锁有两种方式，只要事务结束，锁就会自动事务，包括提交事务和结束事务。

我们也来验证一下，看看共享锁是不是可以重复获取。

Transaction 1	Transaction 2
begin;	
SELECT * FROM student WHERE id=1 LOCK IN SHARE MODE;	
	begin;
	SELECT * FROM student WHERE id=1 LOCK IN SHARE MODE; // OK

2.3 排它锁

第二个行级别的锁叫做 Exclusive Locks（排它锁），它是用来操作数据的，所以又叫做写锁。只要一个事务获取了一行数据的排它锁，其他的事务就不能再获取这一行数据的共享锁和排它锁。

排它锁的加锁方式有两种，第一种是自动加排他锁。我们在操作数据的时候，包括增删改，都会默认加上一个排它锁。

还有一种是手工加锁，我们用一个 FOR UPDATE 给一行数据加上一个排它锁，这个无论是在我们的代码里面还是操作数据的工具里面，都比较常用。

释放锁的方式跟前面是一样的。

排他锁的验证：

Transaction 1	Transaction 2
begin;	
UPDATE student SET sname = '猫老公 555' WHERE id=1;	
	begin;

	SELECT * FROM student WHERE id=1 LOCK IN SHARE MODE; // BLOCKED
	SELECT * FROM student where id=1 FOR UPDATE; // BLOCKED
	DELETE FROM student where id=1 ; // BLOCKED

这个是两个行锁，接下来就是两个表锁。

2.4 意向锁

意向锁是什么呢？我们好像从来没有听过，也从来没有使用过，其实他们是由数据库自己维护的。

也就是说，当我们给一行数据加上共享锁之前，数据库会自动在这张表上面加一个意向共享锁。

当我们给一行数据加上排他锁之前，数据库会自动在这张表上面加一个意向排他锁。

反过来说：

如果一张表上面至少有一个意向共享锁，说明有其他的事务给其中的某些数据行加上了共享锁。

如果一张表上面至少有一个意向排他锁，说明有其他的事务给其中的某些数据行加上了排他锁。

```
select * from t2   where id =4 for update;
```

```
TABLE LOCK table `gupao`.`t2` trx id 24467 lock mode IX
RECORD LOCKS space id 64 page no 3 n bits 72 index PRIMARY of table `gupao`.`t2` trx id 24467 lock_mode X locks rec but not gap
```

那么这两个表级别的锁存在的意义是什么呢？第一个，我们有了表级别的锁，在 InnoDB 里面就可以支持更多粒度的锁。它的第二个作用，我们想一下，如果说没有意向锁的话，当我们准备给一张表加上表锁的时候，我们首先要做什么？是不是必须先要去判断有没其他的事务锁定了其中了某些行？如果有的话，肯定不能加上表锁。那么这个时候我们就要去扫描整张表才能确定能不能成功加上一个表锁，如果数据量特别大，比如有上千万的数据的时候，加表锁的效率是不是很低？

但是我们引入了意向锁之后就不一样了。我只要判断这张表上面有没有意向锁，如果有，就直接返回失败。如果没有，就可以加锁成功。所以 InnoDB 里面的表锁，我们可以把它理解成一个标志。就像火车上厕所有没有人使用的灯，是用来提高加锁的效率的。

Transaction 1	Transaction 2
begin;	
SELECT * FROM student where id=1 FOR UPDATE;	
	BEGIN;

	LOCK TABLES student WRITE; // BLOCKED
	UNLOCK TABLES; // 释放表锁的方式

以上就是 MySQL 里面的 4 种基本的锁的模式，或者叫做锁的类型。

到这里我们要思考两个问题，首先，锁的作用是什么？它跟 Java 里面的锁是一样的，是为了解决资源竞争的问题，Java 里面的资源是对象，数据库的资源就是数据表或者数据行。

所以锁是用来解决事务对数据的并发访问的问题的。

那么，锁到底锁住了什么呢？

当一个事务锁住了一行数据的时候，其他的事务不能操作这一行数据，那它到底是锁住了这一行数据，还是锁住了这一个字段，还是锁住了别的什么东西呢？

3 行锁的原理

3.1 没有索引的表（假设锁住记录）

首先我们有三张表，一张没有索引的 t1，一张有主键索引的 t2，一张有唯一索引的 t3。

我们先假设 InnoDB 的锁锁住了是一行数据或者一条记录。

我们先来看一下 t1 的表结构 ,它有两个字段 ,int 类型的 id 和 varchar 类型的 name。里面有 4 条数据，1、2、3、4。

Transaction 1	Transaction 2
begin;	
SELECT * FROM t1 WHERE id =1 FOR UPDATE;	
	select * from t1 where id=3 for update; //blocked
	INSERT INTO `t1` (`id`, `name`) VALUES (5, '5'); //blocked

现在我们在两个会话里面手工开启两个事务。

在第一个事务里面，我们通过 where id =1 锁住第一行数据。

在第二个事务里面，我们尝试给 id=3 的这一行数据加锁，大家觉得能成功吗？

【互动】觉得能成功刷 1，觉得不能成功的刷 0。

很遗憾，我们看到红灯亮起，这个加锁的操作被阻塞了。这就有点奇怪了，第一个事务锁住了 id=1 的这行数据，为什么我不能操作 id=3 的数据呢？

我们再来操作一条不存在的数据，插入 id=5。它也被阻塞了。实际上这里整张表都被锁住了。所以，我们的第一个猜想被推翻了，InnoDB 的锁锁住的应该不是 Record。

那为什么在没有索引或者没有用到索引的情况下，会锁住整张表？这个问题我们先留在这里。

我们继续看第二个演示。

3.2 有主键索引的表

我们看一下 t2 的表结构。字段是一样的，不同的地方是 id 上创建了一个主键索引。里面的数据是 1、4、7、10。

Transaction 1	Transaction 2
begin;	
select * from t2 where id=1 for update;	
	select * from t2 where id=1 for update; //

	blocked
	select * from t2 where id=4 for update; // OK

第一种情况，使用相同的 id 值去加锁，冲突；使用不同的 id 加锁，可以加锁成功。那么，既然不是锁定一行数据，有没有可能是锁住了 id 的这个字段呢？

3.3 唯一索引（假设锁住字段）

我们看一下 t3 的表结构。字段还是一样的， id 上创建了一个主键索引，name 上创建了一个唯一索引。里面的数据是 1、4、7、10。

Transaction 1	Transaction 2
begin;	
select * from t3 where name= '4' for update;	
	select * from t3 where name = '4' for update; // blocked
	select * from t3 where id = 4 for update; // blocked

在第一个事务里面，我们通过 name 字段去锁定值是 4 的这行数据。

在第二个事务里面，尝试获取一样的排它锁，肯定是失败的，这个不用怀疑。

在这里我们怀疑 InnoDB 锁住的是字段，所以这次我换一个字段，用 id=4 去给这行数据加锁，大家觉得能成功吗？

【互动】觉得能成功的刷一波 1，觉得不能成功的刷一波 0。

很遗憾，又被阻塞了，说明锁住的是字段的这个推测也是错的，否则就不会出现第一个事务锁住了 name，第二个字段锁住 id 失败的情况。

既然锁住的不是 record，也不是 column，InnoDB 里面锁住的到底是什么呢？在这三个案例里面，我们要去分析一下他们的差异在哪里，也就是这三张表的结构，是什么区别导致了加锁的行为的差异？其实答案就是索引。InnoDB 的行锁，就是通过锁住索引来实现的。

那索引又是个什么东西？为什么它可以被锁住？我们在第二节课里面已经分析过了。

那么我们还有两个问题没有解决：

1、为什么表里面没有索引的时候，锁住一行数据会导致锁表？

或者说，如果锁住的是索引，一张表没有索引怎么办？

所以，一张表有没有可能没有索引？

1）如果我们定义了主键(PRIMARY KEY)，那么 InnoDB 会选择主键作为聚集索引。

2）如果没有显式定义主键，则 InnoDB 会选择第一个不包含有 NULL 值的唯一索引作为主键索引。

3）如果也没有这样的唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐藏的聚集索引，它会随着行记录的写入而主键递增。

所以，为什么锁表，是因为查询没有使用索引，会进行全表扫描，然后把每一个隐藏的聚集索引都锁住了。

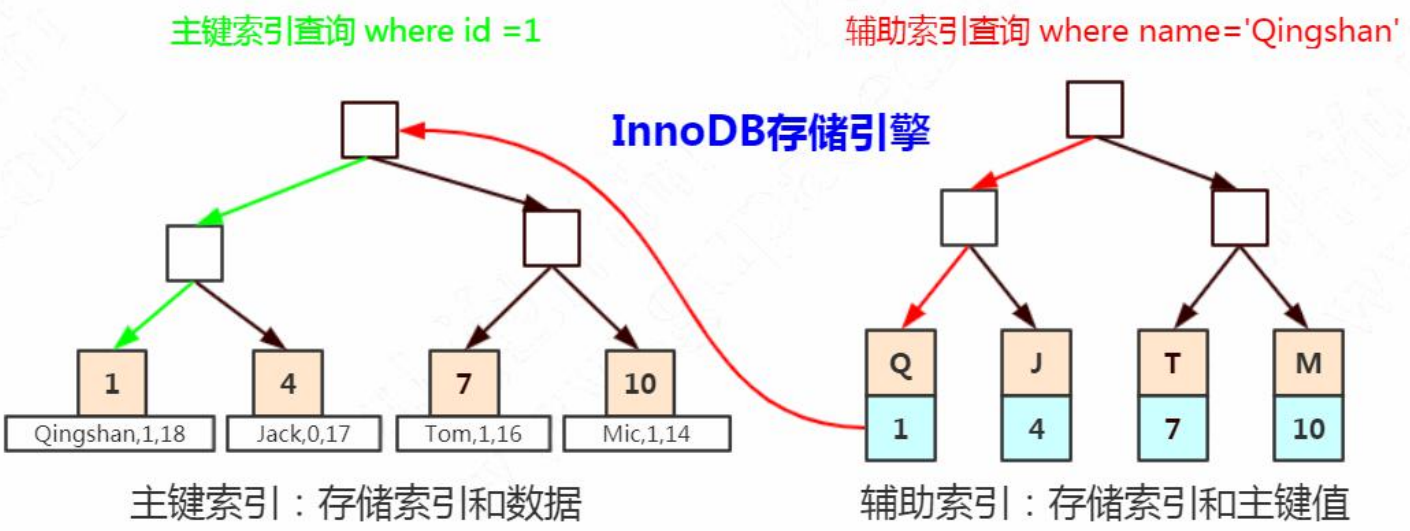
2、为什么通过唯一索引给数据行加锁，主键索引也会被锁住？

大家还记得在 InnoDB 里面，当我们使用辅助索引的时候，它是怎么检索数据的吗？辅助索引的叶子节点存储的是什么内容？

在辅助索引里面，索引存储的是二级索引和主键的值。比如 name=4，存储的是 name 的索引和主键 id 的值 4。

而主键索引里面除了索引之外，还存储了完整的数据。所以我们通过辅助索引锁定一行数据的时候，它跟我们检索数据的步骤是一样的，会通过主键值找到主键索引，然

后也锁定。



现在我们已经搞清楚 4 个锁的基本类型和锁的原理了，在官网上，还有 3 种锁，我们把它理解为锁的算法。我们也来看下 InnoDB 在什么时候分别锁住什么范围。

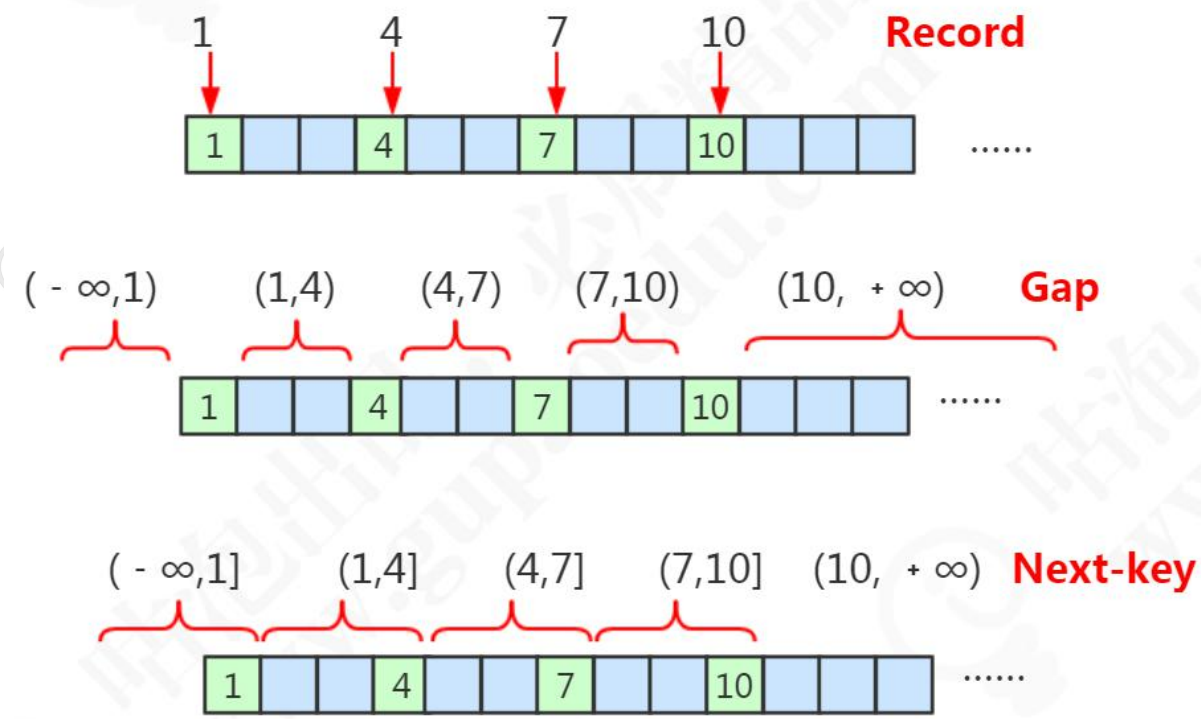
4 锁的算法

我们先来看一下我们测试用的表，t2，这张表有一个主键索引。

我们插入了 4 行数据，主键值分别是 1、4、7、10。

为了让大家真正理解这三种行锁算法的区别，我们需要了解一下三种范围的概念。

因为我们用主键索引加锁，我们这里的划分标准就是主键索引的值。



这些数据库里面存在的主键值，我们把它叫做 Record，记录，那么这里我们就有 4 个 Record。

根据主键，这些存在的 Record 隔开的数据不存在的区间，我们把它叫做 Gap，间隙，它是一个左开右开的区间。

最后一个，间隙（Gap）连同它左边的记录（Record），我们把它叫做临键的区间，它是一个左开右闭的区间。

t2 的主键索引，它是整型的，可以排序，所以才有这种区间。如果我的主键索引不是整形，是字符怎么办呢？字符可以排序吗？ 用 ASCII 码来排序。

我们已经弄清楚了三个范围的概念，下面我们就来看一下在不同的范围下，行锁是怎么表现的。

4.1 记录锁

第一种情况，当我们对于唯一性的索引（包括唯一索引和主键索引）使用等值查询，

精准匹配到一条记录的时候，这个时候使用的就是记录锁。

比如 where id = 1　4　7　10 。

这个演示我们在前面已经看过了。我们使用不同的 key 去加锁，不会冲突，它只锁住这个 record。

4.2 间隙锁

第二种情况，当我们查询的记录不存在，没有命中任何一个 record，无论是用等值查询还是范围查询的时候，它使用的都是间隙锁。

举个例子，where id >4 and id <7，where id = 6。

Transaction 1	Transaction 2
begin;	

	INSERT INTO `t2` (`id`, `name`) VALUES (5, '5'); // BLOCKED
	INSERT INTO `t2` (`id`, `name`) VALUES (6, '6'); // BLOCKED
	select * from t2　where id =6 for update; // OK

select * from t2 where id >20 for update;	
	INSERT INTO `t2` (`id`, `name`) VALUES (11, '11'); // BLOCKED

重复一遍，当查询的记录不存在的时候，使用间隙锁。

注意，间隙锁主要是阻塞插入 insert。相同的间隙锁之间不冲突。

Gap Lock 只在 RR 中存在。如果要关闭间隙锁，就是把事务隔离级别设置成 RC，并且把 innodb_locks_unsafe_for_binlog 设置为 ON。

这种情况下除了外键约束和唯一性检查会加间隙锁，其他情况都不会用间隙锁。

4.3 临键锁

第三种情况，当我们使用了范围查询，不仅仅命中了 Record 记录，还包含了 Gap 间隙，在这种情况下我们使用的就是临键锁，它是 MySQL 里面默认的行锁算法，相当于记录锁加上间隙锁。

其他两种退化的情况：

唯一性索引，等值查询匹配到一条记录的时候，退化成记录锁。

没有匹配到任何记录的时候，退化成间隙锁。

比如我们使用>5 <9， 它包含了记录不存在的区间，也包含了一个 Record 7。

Transaction 1	Transaction 2
begin;	
select * from t2 where id >5 and id < 9 for update;	
	begin;

	select * from t2 where id =4 for update; // OK
	INSERT INTO `t2` (`id`, `name`) VALUES (6, '6'); // BLOCKED
	INSERT INTO `t2` (`id`, `name`) VALUES (8, '8'); // BLOCKED
	select * from t2 where id =10 for update; // BLOCKED

临键锁，锁住最后一个 key 的下一个左开右闭的区间。

```
select * from t2 where id >5 and id <=7 for update;  -- 锁住(4,7]和(7,10]
select * from t2 where id >8 and id <=10 for update;  -- 锁住 (7,10]，(10,+∞)
```

为什么要锁住下一个左开右闭的区间？——就是为了解决幻读的问题。

4.4 小结：隔离级别的实现

所以，我们再回过头来看下这张图片，为什么 InnoDB 的 RR 级别能够解决幻读的问题，就是用临键锁实现的。

我们再回过头来看下这张图片 这个就是 MySQL InnoDB 里面事务隔离级别的实现。

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对 InnoDB 不可能
串行化（Serializable）	不可能	不可能	不可能

最后我们来总结一下四个事务隔离级别的实现：

4.4.1 Read Uncommited

RU 隔离级别：不加锁。

4.4.2 Serializable

Serializable 所有的 select 语句都会被隐式的转化为 select ... in share mode，会和 update、delete 互斥。

这两个很好理解，主要是 RR 和 RC 的区别？

4.4.3 Repeatable Read

RR 隔离级别下，普通的 select 使用快照读(snapshot read)，底层使用 MVCC 来实现。

加锁的 select(select ... in share mode / select ... for update)以及更新操作 update, delete 等语句使用当前读（current read），底层使用记录锁、或者间隙锁、临键锁。

4.4.4 Read Commited

RC 隔离级别下，普通的 select 都是快照读，使用 MVCC 实现。

加锁的 select 都使用记录锁，因为没有 Gap Lock。

除了两种特殊情况——外键约束检查(foreign-key constraint checking)以及重复键检查(duplicate-key checking)时会使用间隙锁封锁区间。

所以 RC 会出现幻读的问题。

5 事务隔离级别怎么选？

https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html

RU 和 Serializable 肯定不能用。为什么有些公司要用 RC，或者说网上有些文章推荐有 RC？

RC 和 RR 主要有几个区别：

1、 RR 的间隙锁会导致锁定范围的扩大。

2、 条件列未使用到索引，RR 锁表，RC 锁行。

3、 RC 的“半一致性”（semi-consistent）读可以增加 update 操作的并发性。

在 RC 中，一个 update 语句，如果读到一行已经加锁的记录，此时 InnoDB 返回记录最近提交的版本，由 MySQL 上层判断此版本是否满足 update 的 where 条件。若满足(需要更新)，则 MySQL 会重新发起一次读操作，此时会读取行的最新版本(并加锁)。

实际上，如果能够正确地使用锁（避免不使用索引去枷锁），只锁定需要的数据，用默认的 RR 级别就可以了。

在我们使用锁的时候，有一个问题是需要注意和避免的，我们知道，排它锁有互斥的特性。一个事务或者说一个线程持有锁的时候，会阻止其他的线程获取锁，这个时候会造成阻塞等待，如果循环等待，会有可能造成死锁。

这个问题我们需要从几个方面来分析，一个是锁为什么不释放，第二个是被阻塞了怎么办，第三个死锁是怎么发生的，怎么避免。

6 死锁

6.1 锁的释放与阻塞

回顾：锁什么时候释放？

事务结束（commit，rollback）；客户端连接断开。

如果一个事务一直未释放锁，其他事务会被阻塞多久？会不会永远等待下去？如果是，在并发访问比较高的情况下，如果大量事务因无法立即获得所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖跨数据库。

[Err] 1205 - Lock wait timeout exceeded; try restarting transaction

MySQL 有一个参数来控制获取锁的等待时间，默认是 50 秒。

```
show VARIABLES like 'innodb_lock_wait_timeout';
```

Variable_name	Value
innodb_lock_wait_timeout	50

对于死锁，是无论等多久都不能获取到锁的，这种情况，也需要等待 50 秒钟吗？那不是白白浪费了 50 秒钟的时间吗？

我们先来看一下什么时候会发生死锁。

6.2 死锁的发生和检测

死锁演示：

Session 1	Session 2

begin; select * from t2 where id =1 for update;	
	begin; delete from t2 where id =4 ;
update t2 set name= '4d' where id =4 ;	
	delete from t2 where id =1 ;

在第一个事务中，检测到了死锁，马上退出了，第二个事务获得了锁，不需要等待 50 秒：

[Err] 1213 - Deadlock found when trying to get lock; try restarting transaction

为什么可以直接检测到呢？是因为死锁的发生需要满足一定的条件，所以在发生死锁时，InnoDB 一般都能通过算法（wait-for graph）自动检测到。

那么死锁需要满足什么条件？死锁的产生条件：

因为锁本身是互斥的，（1）同一时刻只能有一个事务持有这把锁，（2）其他的事务需要在这个事务释放锁之后才能获取锁，而不可以强行剥夺，（3）当多个事务形成等待环路的时候，即发生死锁。

举例：

理发店有两个总监。一个负责剪头的 Tony 总监，一个负责洗头的 Kelvin 总监。

Tony 不能同时给两个人剪头，这个就叫互斥。

Tony 在给别人在剪头的时候，你不能让他停下来帮你剪头，这个叫不能强行剥夺。

如果 Tony 的客户对 Kelvin 总监说 :你不帮我洗头我怎么剪头？Kelvin 的客户对 Tony 总监说：你不帮我剪头我怎么洗头？这个就叫形成等待环路。

如果锁一直没有释放，就有可能造成大量阻塞或者发生死锁，造成系统吞吐量下降，这时候就要查看是哪些事务持有了锁。

6.3 查看锁信息（日志）

SHOW STATUS 命令中，包括了一些行锁的信息：

```
show status like 'innodb_row_lock_%';
```

Variable_name	Value
Innodb_row_lock_current_waits	0
Innodb_row_lock_time	0
Innodb_row_lock_time_avg	0
Innodb_row_lock_time_max	0
Innodb_row_lock_waits	0

非实时

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的总时间长度，单位 ms；
Innodb_row_lock_time_avg ： 每次等待所花平均时间；
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 ： 从系统启动到现在总共等待的次数。

SHOW 命令是一个概要信息。InnoDB 还提供了三张表来分析事务与锁的情况：

```
select * from information_schema.INNODB_TRX;  -- 当前运行的所有事务 ， 还有具体的语句
```

trx_id	trx_state	trx_started	trx_requested_lock_id	trx_wait_started	trx_weight	trx_mysql_thread_id	trx_query	trx_operation_state
25386	RUNNING	2019-12-12 23:49:22	(Null)	(Null)	1	4	(Null)	(Null)
25385	RUNNING	2019-12-12 23:49:07	(Null)	(Null)	2	8	(Null)	(Null)
25348	RUNNING	2019-12-12 23:44:50	(Null)	(Null)	2	7	(Null)	(Null)

```
select * from information_schema.INNODB_LOCKS;  -- 当前出现的锁
```

lock_id	lock_trx_id	lock_mode	lock_type	lock_table	lock_index	lock_space	lock_page	lock_rec	lock_data
25385:64:3:3	25385	X	RECORD	`gupao`.`t2`	PRIMARY	64	3	3	4
25348:64:3:3	25348	X	RECORD	`gupao`.`t2`	PRIMARY	64	3	3	4

```
select * from information_schema.INNODB_LOCK_WAITS;  -- 锁等待的对应关系
```

requesting_trx_id	requested_lock_id	blocking_trx_id	blocking_lock_id
25386	25386:64:3:4	25385	25385:64:3:4

找出持有锁的事务之后呢？

如果一个事务长时间持有锁不释放，可以 kill 事务对应的线程 ID，也就是 INNODB_TRX 表中的 trx_mysql_thread_id，例如执行 kill 4，kill 7，kill 8。

当然，死锁的问题不能每次都靠 kill 线程来解决，这是治标不治本的行为。我们应该尽量在应用端，也就是在编码的过程中避免。

有哪些可以避免死锁的方法呢？

6.4 死锁的避免

1、 在程序中，操作多张表时，尽量以相同的顺序来访问（避免形成等待环路）；

2、 批量操作单张表数据的时候，先对数据进行排序（避免形成等待环路）；

3、 申请足够级别的锁，如果要操作数据，就申请排它锁；

4、 尽量使用索引访问数据，避免没有 where 条件的操作，避免锁表；

5、 如果可以，大事务化成小事务；

6、 使用等值查询而不是范围查询查询数据，命中记录，避免间隙锁对并发的影响。

集中答疑帖：

https://gper.club/articles/7e7e7f7ff7g55gc8g6c

作者：咕泡学院-青山

最后更新时间：2020 年 1 月 4 日 22:55:59