

MYSQL 事务/事务ACID特性/事务隔离级别/锁/MVCC

1.事务

1.1什么是事务:

数据库操作的最小工作单元，是作为单个逻辑工作单元执行的一系列操作；

事务是一组不可再分割的操作集合（工作逻辑单元）；

1.2典型事务场景(转账):

```
update user_account set balance = balance - 1000 where userID = 3;  
update user_account set balance = balance +1000 where userID = 1;
```

1.3mysql中如何开启事务:

```
begin / start transaction      -- 手工  
commit / rollback             -- 事务提交或回滚  
set session autocommit = on/off; -- 设定事务是否自动开启
```

1.4mysql默认事务级别为：自动提交

```
select VARIABLES like 'autocommit';
```

结果为：on，表示自动提交。

1.5JDBC 编程:

connection.setAutoCommit (boolean) ;

1.6Spring 事务AOP编程:

expression=execution (com.gpedu.dao(..))

2.事务ACID特性

- 原子性 (Atomicity)
 - 最小的工作单元，整个工作单元要么一起提交成功，要么全部失败回滚
- 一致性 (Consistency)
 - 事务中操作的数据及状态改变是一致的，即写入资料的结果必须完全符合预设的规则，不会因为出现系统意外等原因导致状态的不一致
- 隔离性 (Isolation)
 - 一个事务所操作的数据在提交之前，对其他事务的可见性设定（一般设定为不可见）
- 持久性 (Durability) 事务所做的修改就会永久保存，不会因为系统意外导致数据的丢失

3.事务隔离级别

[SQL92 ANSI/ISO标准:](#)

事务隔离级别:

- Read Uncommitted (未提交读) --未解决并发问题

事务未提交对其他事务也是可见的, 脏读 (dirty read)

- Read Committed (提交读) --解决脏读问题

一个事务开始之后, 只能看到自己提交的事务所做的修改, 不可重复读 (nonrepeatable read)

- Repeatable Read (可重复读) --解决不可重复读问题

在同一个事务中多次读取同样的数据结果是一样的, 这种隔离级别未定义解决幻读的问题

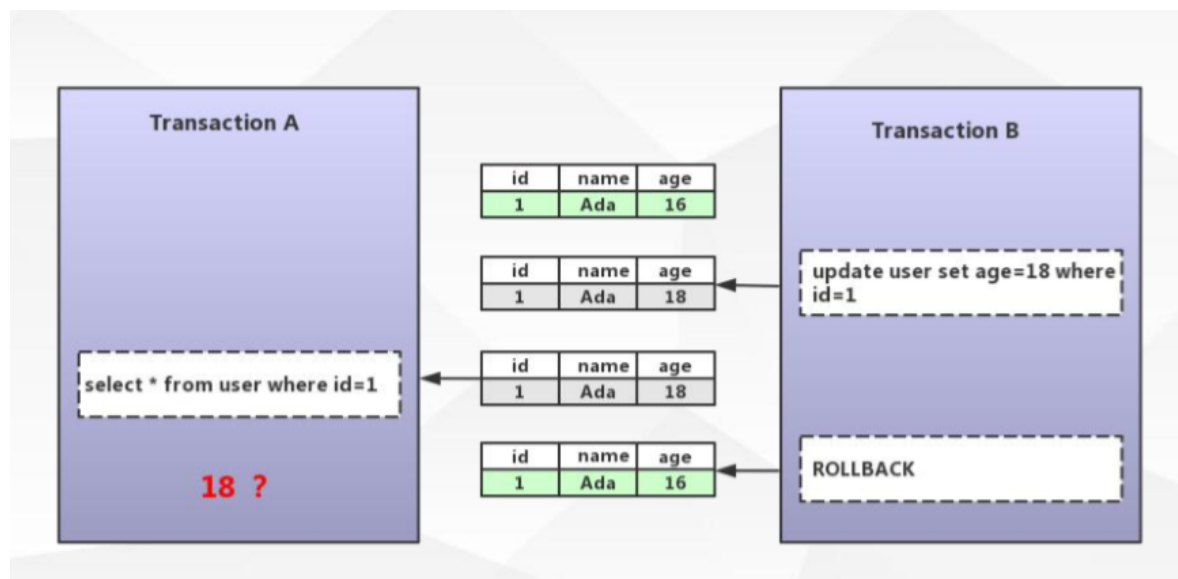
- Serializable (串行化) --解决所有问题

最高的隔离级别, 通过强制事务的串行执行

事务并发带来了哪些问题

3.1 Read Uncommitted (未提交读)

图一



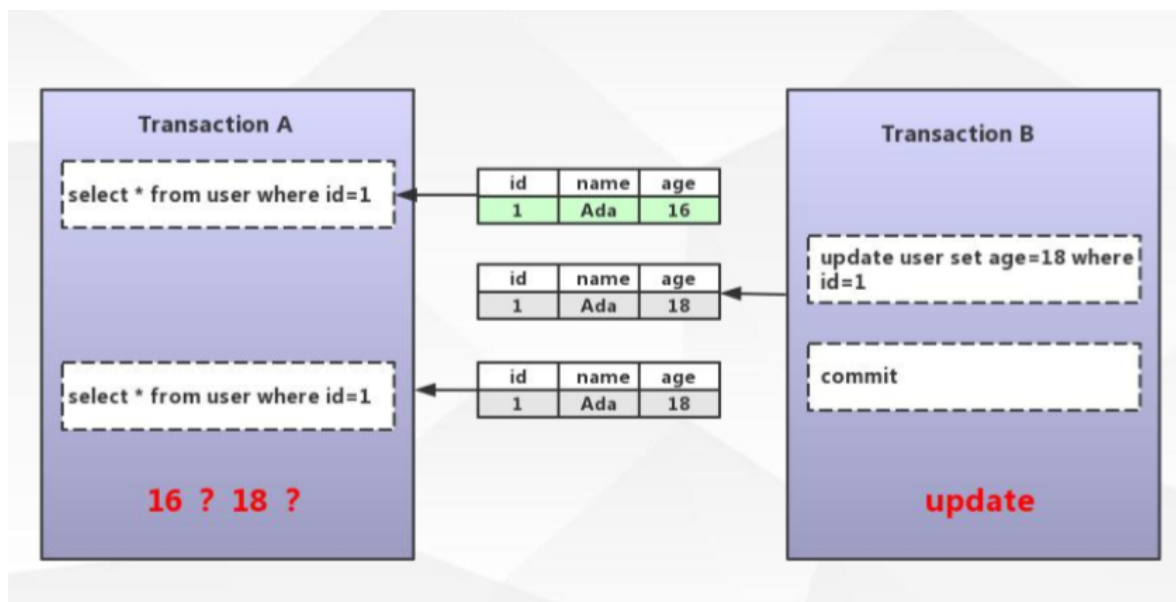
事务B先update, 接着事务A select, 然后事务B roolaback。事务A产生脏读现象。

解决:

通过将事务隔离级别【Read Uncommitted(为提交读)】--> Read Committed (提交读), 即可解决图一的问题。

3.2Read Committed (提交读)

图二



事务A先select，接着事务B update，然后事务B commit，然后事务A select。事务A在一次开启事务的过程中，两次查询，得到两次不同的数据，产生可重复度的问题。

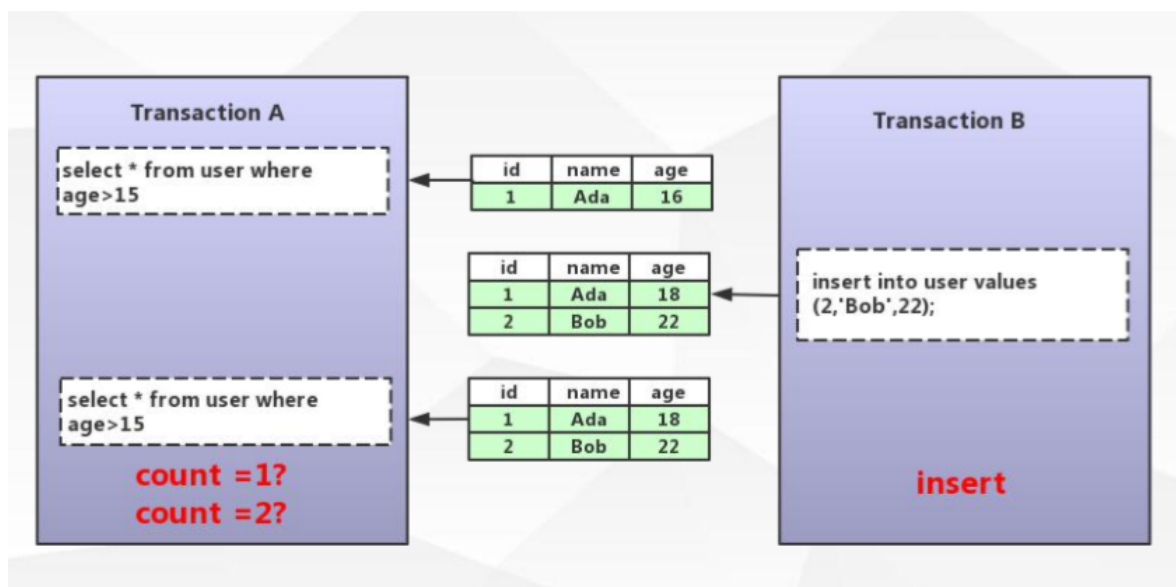
解决：

通过将事务隔离级别：Read Committed（提交读）--> Repeatable Read（可重复读）。

即可解决图二的问题。

3.3 Repeatable Read（可重复读）

图三



事务A先select，接着事务B insert 但未commit，然后事务A select。事务A在一次开启事务的过程中，两次查询的结果集不同，产生幻读的问题。

解决：

通过将事务隔离级别：Repeatable Read（可重复读）--> Serializable（串行化）。

即可解决图三的问题。

3.4 InnoDB引擎对隔离级别的支持程度

事务隔离级别的并发能力由串行化-->未提交读 越来越高

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



4.锁

锁是用于管理不同事务对共享资源的并发访问

表锁与行锁的区别：

锁定粒度：表锁 > 行锁

加锁效率：表锁 > 行锁

冲突概率：表锁 > 行锁

并发性能：表锁 < 行锁

4.1 MySQL InnoDB锁类型

- 共享锁（行锁）：Shared Locks
- 排它锁（行锁）：Exclusive Locks
- 意向锁共享锁（表锁）：Intention Shared Locks
- 意向锁排它锁（表锁）：Intention Exclusive Locks
- 自增锁：AUTO-INC Locks

行锁的算法

- 记录锁 Record Locks
- 间隙锁 Gap Locks
- 临键锁 Next-key Locks 行锁的算法

4.1.1 共享锁 (Shared Locks) vs 排它锁 (Exclusive Lock)

共享锁：

- 又称为读锁，简称S锁，顾名思义，共享锁就是多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改；
- 加锁释锁方式：
 - `select * from users WHERE id=1 LOCK IN SHARE MODE;`
 - `commit/rollback`

排他锁：

- 又称为写锁，简称X锁，排他锁不能与其他锁并存，如一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的锁（共享锁、排他锁），只有该获取了排他锁的事务是可以对数据行进行读取和修改，（其他事务要读取数据可来自于快照）

排它锁，不影响另一个事务的 `select * from table` 。但影响另一个事务的操作：`delete / update / insert / SELECT * FROM table_name WHERE ... FOR UPDATE` ，不能同时执行。

加锁释放锁方式：

`delete / update / insert` 默认加上X锁

`SELECT * FROM table_name WHERE ... FOR UPDATE` 也相当于加上X锁

`commit/rollback`

InnoDB 行锁到底锁了什么

- InnoDB的行锁是通过给索引上的索引项加锁来实现的。
- 只有通过索引条件进行数据检索，InnoDB才使用行级锁，否则，InnoDB 将使用表锁（锁住索引的所有记录）（另类的行锁）

如上述两点，我们在删除/修改数据时，一定要让where条件命中我们的索引，否则，会锁表，影响别的事务对此表的操作。

表锁：`lock tables xx read/write;`

4.1.2意向锁共享锁 (IS) &意向锁排它锁 (IX)

意向锁(IS、IX) 是（表锁）

- 意向共享锁(IS)
 - 表示事务准备给数据行加入共享锁，即一个数据行加共享锁前必须先取得该表的IS锁，意向共享锁之间是可以相互兼容的
- 意向排它锁(IX)
 - 表示事务准备给数据行加入排他锁，即一个数据行加排他锁前必须先取得该表的IX锁，意向排它锁之间是可以相互兼容的

意向锁(IS、IX)是InnoDB数据操作之前自动加的，不需要用户干预。

意向锁(IS、IX)意义：

当事务想去进行锁表时，可以先判断意向锁是否存在，存在时则可快速返回该表不能启用表锁

4.1.3自增锁AUTO-INC Locks

针对自增列自增长的一个特殊的表级别锁

```
show variables like 'innodb_autoinc_lock_mode';
```

默认取值1，代表连续，事务未提交ID永久丢失。

4.1.4记录锁 (Record) &间隙锁 (Gap) &临建锁 (Next-key)

- Next-key locks:
 - 锁住记录+区间（左开右闭）

当sql执行按照索引进行数据的检索时,查询条件为范围查找 (between and、<、>等【含义就是 explain 执行计划命中的索引类型为 range类型】) 并有数 据命中则此时SQL语句加上的锁为Next-key locks，锁住索引的记录+区间 (左开右闭)

【将表中数据每行之间当作一个区间，如果有4条数据，则会生成5个区间，如下图】



- **Gap locks:**

- 锁住数据不存在的区间 (左开右开)

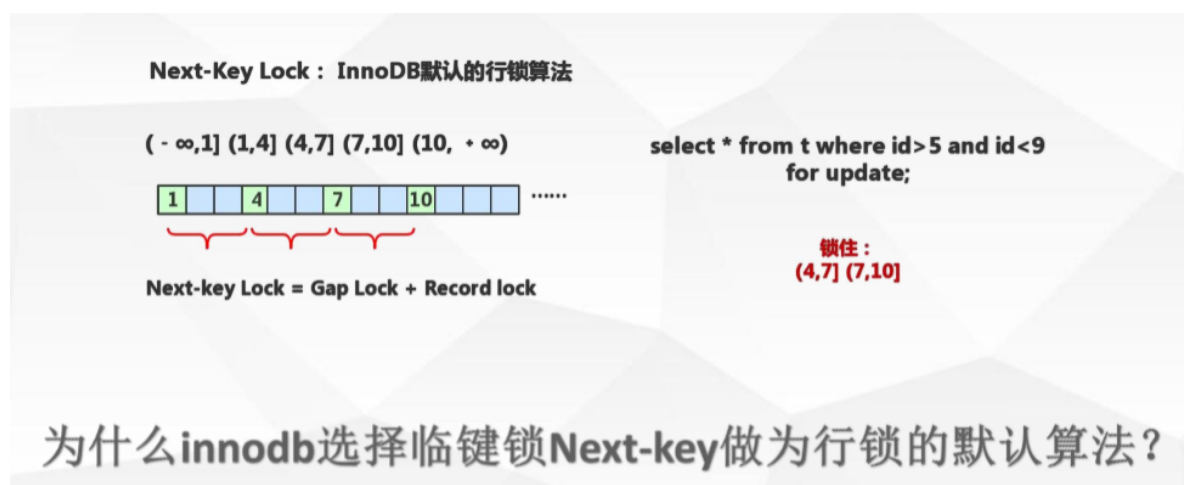
当sql执行按照索引进行数据的检索时，查询条件的数据不存在，这时SQL语句加上的锁即为 Gap locks，锁住索引不存在的区间 (左开右开)

- **Record locks:**

- 锁住具体的索引项

当sql执行按照唯一性 (Primary key、Unique key) 索引进行数据的检索时，查询条件等值匹配且查询的数据是存在，这时SQL语句加上的锁即为记录锁Record locks，锁住具体的索引项

4.1.5临建锁 (Next-key)



上图，表结构： id, name; 且id无自增，id为主键。

表中只有四条数据：

id	name
1	1
4	4
7	7
10	10

如上图，将表中数据每行之间当作一个区间，如果有4条数据，则会生成5个区间： $(-\infty, 1]$ $(1, 4]$ $(4, 7]$ $(7, 10]$ $(10, +\infty]$

临建锁 大致含义：

操作：

1.事务A 执行 begin；开启事务，

2.事务A `select * from t where id>5 and id<9 for update`；添加X锁（排它锁），**锁住了(4, 7] (7, 10] 两个区间。**

此时，事务A会锁住id为7的行，以及 $4 < id \leq 7$ 区间范围，如果另外一个事务B要写这个范围，事务B是阻塞无法执行的。

同时，事务A还会锁住 $7 < id \leq 10$ 区间范围，这个范围的id对应的行的数据，其他事务B也是无法进行写，直到事务A commit后才能写。

3.事务B 执行 begin；开启事务，

4.`select * from t where id=4 for update` ,表示事务B 要加排它锁，执行结果：事务B 可以执行。【符合左开右闭原则】

5.`select * from t where id=7 for update` ,表示事务B 要加排它锁，执行结果：事务B 阻塞，不可以执行。【符合左开右闭原则】

6.`select * from t where id=10 for update` ,表示事务B 要加排它锁，执行结果：事务B 阻塞，不可以执行。【符合左开右闭原则】

6.`select * from t where id=11 for update` ,表示事务B 要加排它锁，执行结果：事务B 执行成功。【符合左开右闭原则】

为什么innodb选择临建锁Next-key作为行锁的默认算法？

答：防止幻读，因为临建锁锁住了查询记录的左右两条数据的区间id，所以，在同一时间其他的事务则无法插入当前记录的临近数据（因为innodb叶子节点中的数据都是顺序性的，查询记录的临界区间肯定也是自增的，就避免了另外事务进行insert临近数据了）

4.1.6间隙锁（Gap）

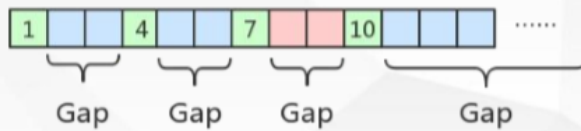
当sql执行按照索引进行数据的检索时，查询条件的数据不存在，这时SQL语句加上的锁即为 **Gap locks**，锁住索引不存在的区间（左开右开）

Gap间隙锁 只在RR事务隔离级别存在

当记录不存在，临键锁退化成Gap锁

Gap Lock：范围查询或等值查询
且记录不存在

$(-\infty, 1)$ $(1, 4)$ $(4, 7)$ $(7, 10)$ $(10, +\infty)$



```
select * from t where id > 4  
and id < 6 for update;
```

```
select * from t where id = 6  
for update;
```

锁住：(4,7)

```
select * from t where id  
> 20 for update;
```

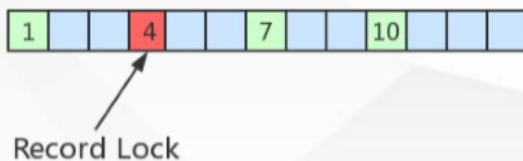
锁住：(10, $+\infty$)

Gap只在RR事务隔离级别存在

4.1.7记录锁 (Record)

唯一性（主键/唯一）索引，条件为精准匹
配，退化成Record锁

Record Lock：唯一索引 unique key(id)
等值查询，精准匹配

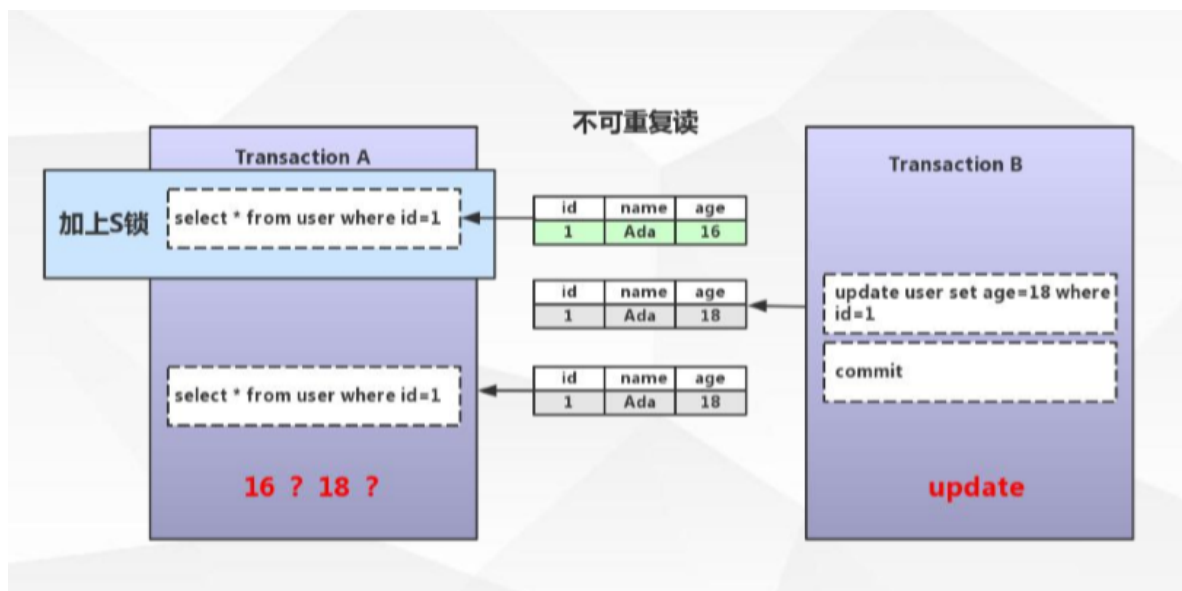


```
select * from t  
where id = 4 for update;
```

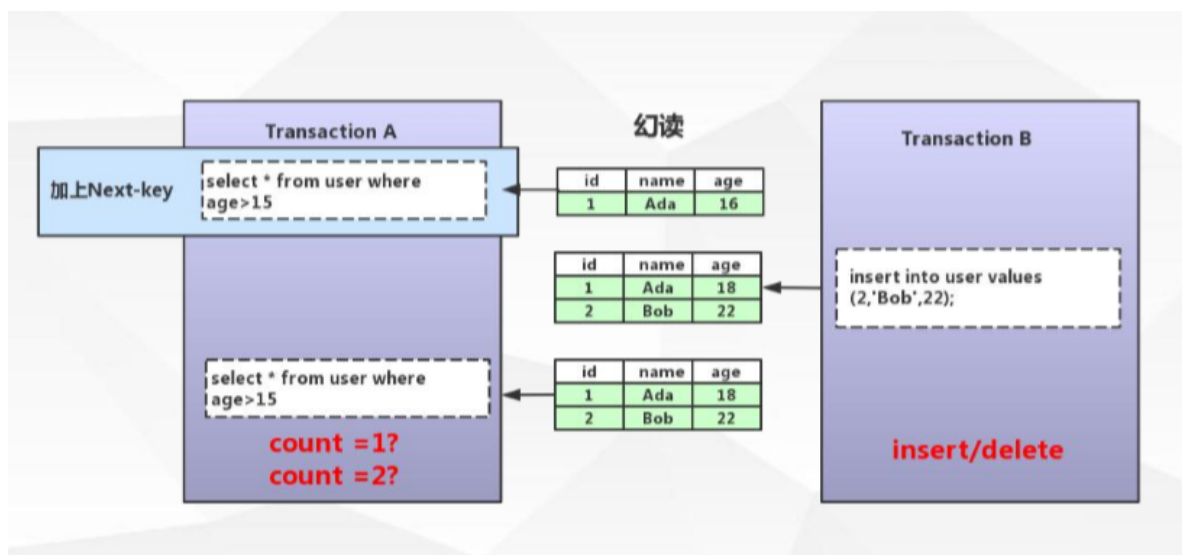
锁住：id=4

1. 可重复的普通索引，查询条件命中4时，锁住的是查询记录的左右两个区间的数据；
2. 只有在唯一性索引当作查询条件命中的时候，才会用到record记录锁，且锁定的行数为当前的唯一记录；

4.2利用锁怎么解决不可重复读



4.3利用锁怎么解决幻读



4.4死锁介绍

- 多个并发事务（2个或者以上）；
- 每个事务都持有锁（或者是已经在等待锁）；
- 每个事务都需要再继续持有锁；
- 事务之间产生加锁的循环等待，形成死锁。

4.5 死锁避免

- 1) 类似的业务逻辑以固定的顺序访问表和行。
- 2) 大事务拆小。大事务更倾向于死锁，如果业务允许，将大事务拆小。
- 3) 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁概率。
- 4) 降低隔离级别，如果业务允许，将隔离级别调低也是较好的选择。
- 5) 为表添加合理的索引。可以看到如果不走索引将会为表的每一行记录添加锁（或者说是表锁）。

4.6 思考引申出MVCC

查看mysql的设置的事务隔离级别
select global.@@tx_isolation; select @@tx_isolation;

ex1:

```
tx1: set session autocommit=off;  
    update users set lastUpdate=now() where id =1;  
    在未做commit/rollback操作之前  
    在其他的事务我们能不能进行对应数据的查询（特别是加上了x锁的数据）  
tx2: select * from users where id > 1;  
    select * from users where id = 1;
```

ex2:

```
tx1: begin  
    select * from users where id =1 ;  
tx2: begin  
    update users set lastUpdate=now() where id =1;  
tx1:  
    select * from users where id =1;
```

这两个案例
从结果上来
看是一致的！

底层实现是
怎样的呢？
是一样的吗？

他们的底层
实现跟MVCC
有什么关系
么？



MVCC

5.MVCC

MVCC: Multiversion concurrency control (多版本并发控制)

普通话解释:

并发访问(读或写)数据库时，对正在事务内处理的数据做 多版本的管理。以达到用来避免写操作的堵塞，从而引发读操作的并发问题。

5.1 Mysql中MVCC逻辑流程-插入

MVCC默认会给数据行加入三列（隐藏的列）：

DB_TRX_ID（数据行版本号），DB_ROLL_PT（删除版本号），ROWNUM（行号）。

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

id	name	age	DB_TRX_ID	DB_ROLL_PT

表: teacher

step: 插入数据

```
假设系统的全局事务ID号从1开始;  
begin;      -- 拿到系统的事务ID=1;  
insert into teacher(name,age) VALUE ('seven',18);  
insert into teacher(name,age) VALUE ('qingshan',19);  
commit;
```

思考:
如果我直接执行
insert into teacher(name,age)
VALUE ('seven',18);
insert into teacher(name,age)
VALUE ('qingshan',19);
且 set autocommit = OFF;

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	NULL

表: teacher

操作:

1.如上，事务A，开启事务，获取全局事务ID号1（事务版本号），插入两条数据，

则在DB_TRX_ID中也存储此事务ID号;DB_ROLL_PT设置为null。

5.2 Mysql中MVCC逻辑流程-删除

表 : teacher

主键 (自增)	姓名	年龄	数据行的版本号	删除版本号
id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	NULL

step : 数据的删除

```
假设系统的全局事务ID号目前到了22
begin;      -- 拿到系统的事务ID=22 ;
delete teacher where id = 2 ;
commit;
```

表 : teacher

主键 (自增)	姓名	年龄	数据行的版本号	删除版本号
id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	22

操作:

2.如上, 事务A, 开启事务, 获取全局事务ID号22 (事务版本号), 删除id=2的数据, 则设置DB_ROLL_PT=22, 表示删除此条记录。

5.3Mysql中MVCC逻辑流程-修改

表 : teacher

主键 (自增)	姓名	年龄	数据行的版本号	删除版本号
id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	22

step : 修改操作

```
假设系统的全局事务ID号目前到了33
begin;      -- 拿到系统的事务ID=33 ;
update teacher set age = 19 where id = 1 ;
commit;
```

表 : teacher

主键 (自增)	姓名	年龄	数据行的版本号	删除版本号
id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	33
2	qingshan	19	1	22
1	seven	19	33	NULL

copy

修改操作是先做命中的数据行的copy, 将原行数据的删除版本号的值设置为当前事务ID(33)

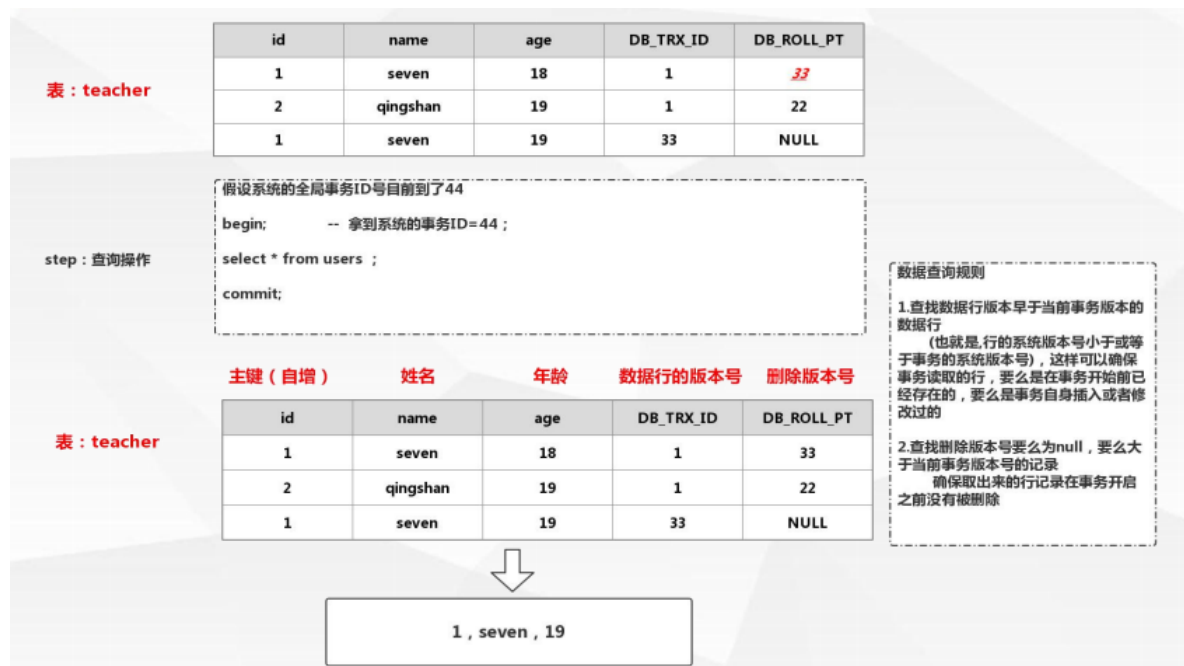
操作:

3.如上, 事务A, 开启事务, 获取全局事务ID号33 (事务版本号), 修改id=1的数据,

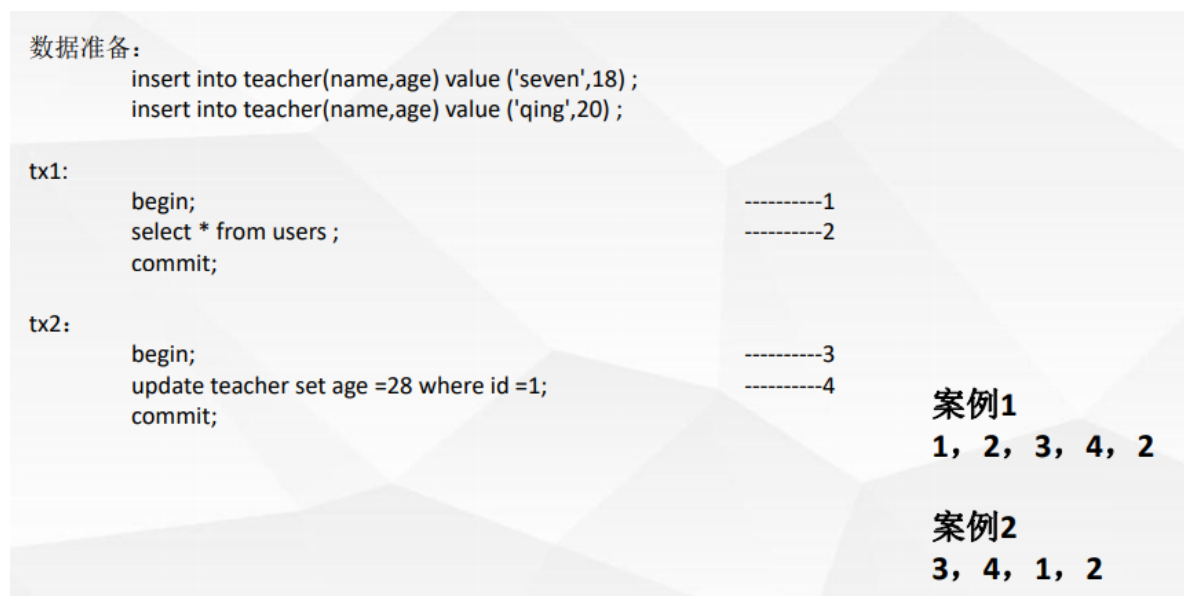
则先copy id=1的数据，然后将之前的数据 DB_ROLL_PT=33，表示事务ID号33操作的这条数据进行写操作。

将新copy的数据：DB_TRX_ID=33,DB_ROLL_PT=NULL。

5.4Mysql中MVCC逻辑流程-查询



5.5Mysql中MVCC版本控制案例



5.5.1MVCC版本控制案例一 (1, 2, 3, 4, 2)

(1, 2, 3, 4, 2) 操作顺序, 属于: a先查询, b修改, a查询, 最终a查询结果不变。

	主键 (自增)	姓名	年龄	数据行的版本号	删除版本号
表 : teacher	id	name	age	DB_TRX_ID	DB_ROLL_PT
	1	seven	18	1	NULL
	2	qingshan	20	1	NULL

1,2 执行之后得到结果 :
TXID =2 进行数据查找

1 , seven , 18
2 , qingshan , 20

3,4 执行之后得到结果 :

copy

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	3
2	qingshan	20	1	NULL
1	seven	28	3	NULL

2执行之后得到的结果 :
TXID =2 进行数据查找

1 , seven , 18
2 , qingshan , 20

5.5.2MVCC版本控制案例一 (3, 4, 1, 2)

	主键 (自增)	姓名	年龄	数据行的版本号	删除版本号
表 : teacher	id	name	age	DB_TRX_ID	DB_ROLL_PT
	1	seven	18	1	NULL
	2	qingshan	20	1	NULL

3,4 执行之后得到结果 :

copy

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	2
2	qingshan	20	1	NULL
1	seven	28	2	NULL

1,2 执行之后得到结果 :
TXID= 3进行数据的查找

1 , seven , 28
2 , qingshan , 20

问题出在哪里 ? ? ? ? ? ? ?

(3, 4, 1, 2) 操作顺序, 属于: a先修改, b查询, b查询结果是a刚修改但未提交的数据, 产生了脏读问题。

问题出在了哪里呢?

6.Undo log

Undo Log是什么?

- undo意为取消, 以撤销操作为目的, 返回指定某个状态的操作
- undo log指事务开始之前, 在操作任何数据之前, 首先将需操作的数据备份到一个地方 (Undo Log)

UndoLog是为了实现事务的原子性而出现的产物

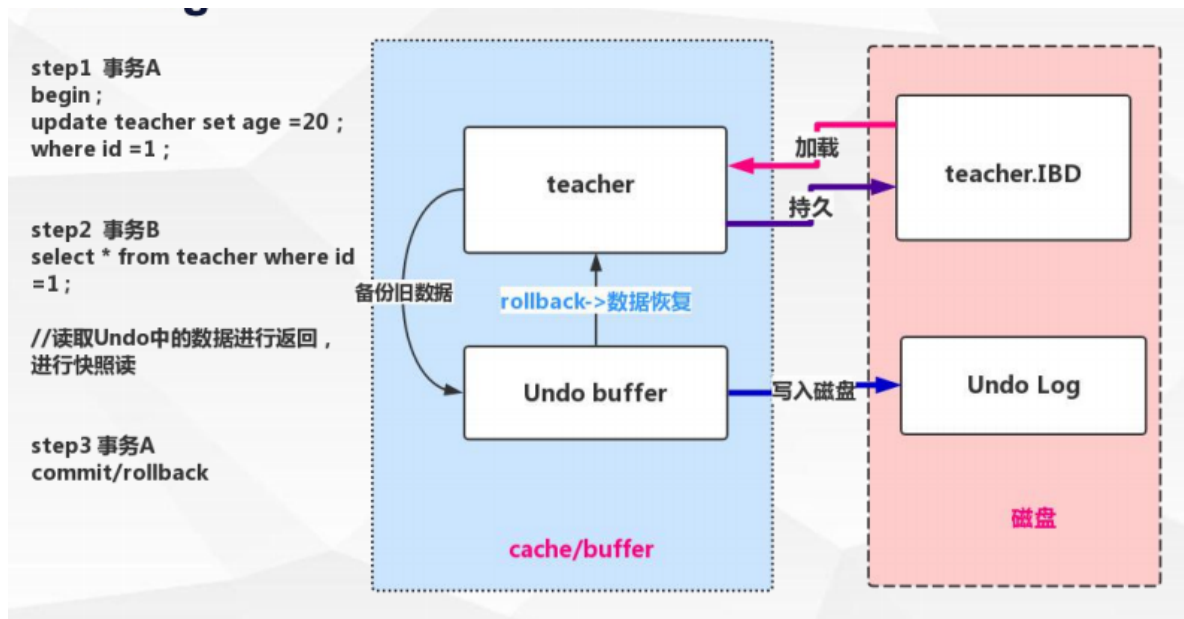
Undo Log实现事务原子性：

- 事务处理过程中如果出现了错误或者用户执行了 ROLLBACK语句,Mysql可以利用Undo Log中的备份 将数据恢复到事务开始之前的状态

UndoLog在Mysql innodb存储引擎中用来实现多版本并发控制

Undo log实现多版本并发控制：

- 事务未提交之前，Undo保存了未提交之前的版本数据，Undo 中的数据可作为数据旧版本快照供其他并发事务进行快照读



6.1当前读/快照读

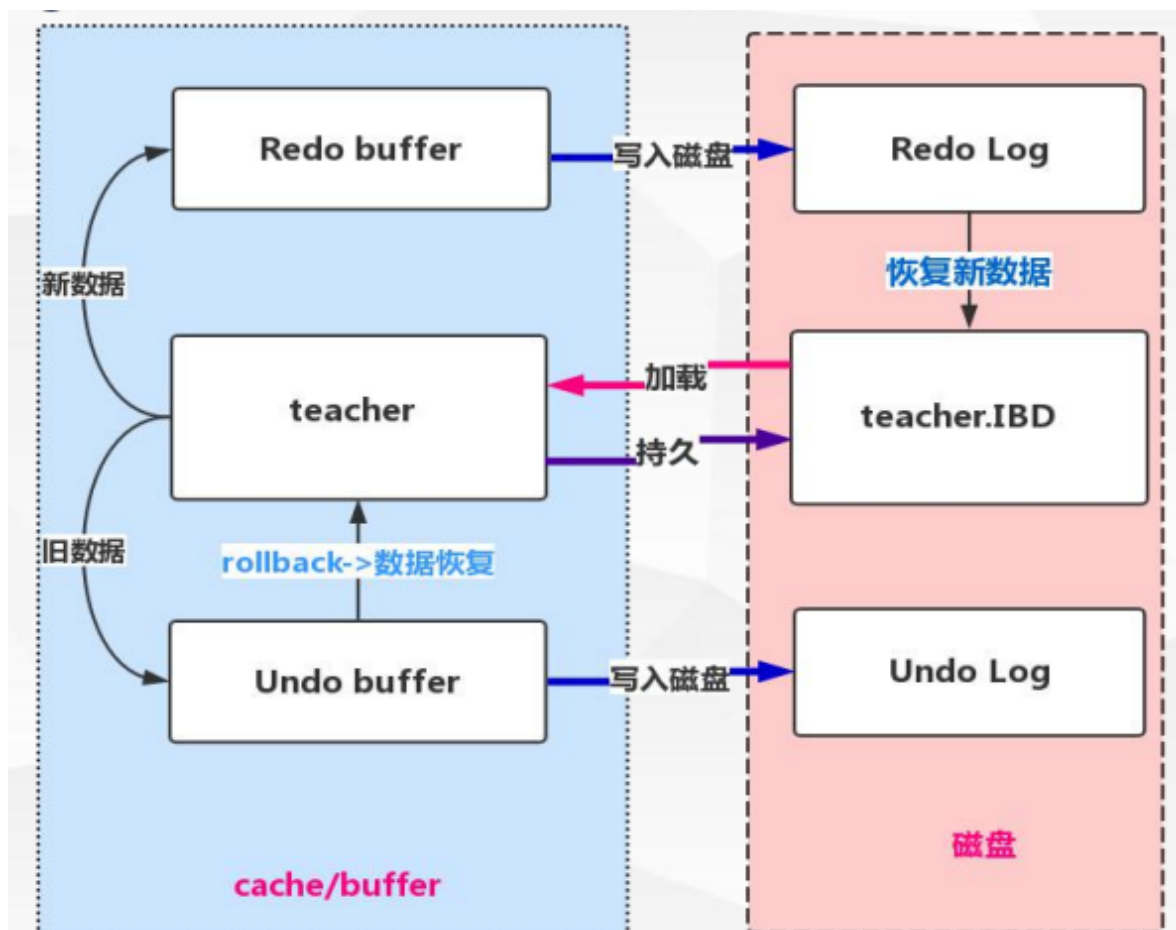
快照读：

- SQL读取的数据是快照版本，也就是历史版本，普通的SELECT就是快照读
- innodb快照读，数据的读取将由 cache(原本数据) + undo(事务修改过的数据) 两部分组成

当前读：

- SQL读取的数据是最新版本。通过锁机制来保证读取的数据无法通过其他事务进行修改
- UPDATE、DELETE、INSERT、SELECT ... LOCK IN SHARE MODE、SELECT ... FOR UPDATE都是当前读

7.Redo Log



Redo Log补充知识点：

- 1.指定Redo log 记录在{datadir}/ib_logfile1&ib_logfile2 可通过innodb_log_group_home_dir 配置指定 目录存储
- 2.一旦事务成功提交且数据持久化落盘之后，此时Redo log中的对应事务数据记录就失去了意义，所以 Redo log的写入是日志文件循环写入的

指定Redo log日志文件组中的数量 innodb_log_files_in_group 默认为2

指定Redo log每一个日志文件最大存储量innodb_log_file_size 默认48M

指定Redo log在cache/buffer中的buffer池大小innodb_log_buffer_size 默认16M

- 3.Redo buffer 持久化Redo log的策略，Innodb_flush_log_at_trx_commit:

取值 0 每秒提交 Redo buffer --> Redo log OS cache -->flush cache to disk[可能丢失一秒内 的事务数据]

取值 1 默认值，每次事务提交执行Redo buffer --> Redo log OS cache -->flush cache to disk [最安全，性能最差的方式]

取值 2 每次事务提交执行Redo buffer --> Redo log OS cache 再每一秒执行 ->flush cache to disk 操作

8.数据库三范式

第一范式（1NF）：

字段具有原子性,不可再分。所有关系型数据库系统都满足第一范式) 数据库表中的字段都是单一属性的，不可再分；

第二范式（2NF）：

要求实体的属性完全依赖于主键。所谓完全依赖是指不能存在仅依赖主键一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，第二范式就是属性完全依赖主键。

第三范式（3NF）：

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主键信息。

简单一点：

- 1，每一列只有一个单一的值，不可再拆分
- 2，每一行都有主键能进行区分
- 3，每一个表都不包含其他表已经包含的非主键信息。