链滴

Mysql Innodb 锁（摘录）

作者：liqitian3344

原文链接：https://ld246.com/article/1573183444400

来源网站：链滴

Mysql Innodb 锁介绍

<table><tr><td bgcolor=orange>简单记录下mysql的东东

1. 为何加锁及事务隔离级别介绍
2. 锁的分类
3. 加锁分析
4. 分析死锁

</td></tr></table>

为什么要加锁

总所周知数据库是一个多用户使用的共享资源。当多个用户并发地存取数据时，在数据库中就会产生个事务同时存取同一数据的情况。若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数库的一致性。

数据库有ACID原则:

1. 原子性(Atomicity):一个事务必须被视为一个不可分割的最小工作单元，整个事务中的所有操作要全部提交成功，要么全部失败回滚，对于一个事务来说，不可能只执行其中的一部分操作，这就是事的原子性
2. 一致性(Consistency): 数据库总是从一个一致性的状态转换到另一个一致性的状态。
3. 持久性(Durability):一旦事务提交，则其所做的修改不会永久保存到数据库。
4. 隔离性(Isolation)：一个事务所做的修改在最终提交以前，对其他事务是不可见的。

其中I是隔离性，标准SQL规范中定义了四种隔离级别：

隔离级别	脏读	不可重复读	幻读
读未提交 Read Uncommitted	√	√	√
读已提交 Read Committed	×	√	√
可重复读 Repeatable Read	×	×	√
串行化 Serializable	×	×	×

通过 show global variables like '%iso%';可以看出mysql默认隔离级别是可重复读.

上图隔离级别越往下，隔离级别越高，问题越少，同时并发度也越低。隔离级别和并发度成反比的。

1. 脏读：事务A读取了事务B未提交的数据
2. 不可重复读：对于一条记录，事务A两次读取的数据变了
3. 幻读：事务A按照相同的查询条件，读取到了新增的数据

隔离级别	脏读	不可重复读	幻读
读未提交 Read Uncommitted	√	√	√
读已提交 Read Committed	×	√	√
可重复读 Repeatable Read	×	×	×
串行化 Serializable	×	×	×

和标准SQL规范相比，MySQL中可重复读解决了脏读，不可能重复度，实现了串行化隔离级别的功能同时没有严重影响并发。是通过加锁、阻止插入新数据，来解决幻读。

通过客户端修改事务隔离级别演示脏读，不可重复度，幻读

准备数据：

```
CREATE TABLE `user` (
 `id` int(11) NOT NULL,
 `name` varchar(20) DEFAULT NULL,
 `sex` varchar(5) DEFAULT NULL,
 PRIMARY KEY (`id`),
 KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

INSERT INTO `test`.`user`(`id`, `name`, `sex`) VALUES (4, 'lala', 'm');
INSERT INTO `test`.`user`(`id`, `name`, `sex`) VALUES (5, 'lala', 'm');
INSERT INTO `test`.`user`(`id`, `name`, `sex`) VALUES (6, 'lala', 'm');
```

开始操作：

分别开启两个mysql客户端设置当前会话隔离级别为读未提交与关闭自动提交

```
set session transaction isolation level read uncommitted;
set autocommit = 0;
```

通过show session variables like '%iso%';

show session variables like 'autocommit';查看是否设置成功

下面操作按中括号[]数字大小顺序进行操作（先执行[0]，在执行[1]，依次类推）

```
#客户端1
#1. 开启事务
start TRANSACTION     [0]
#2. 查看数据
select * from  user       [1]

update user set sex ='123' where id =4       [4]
INSERT into user(id,name,sex) values (7,'lala','m')       [8]
```

```
delete from user where id = 5          [6]
COMMIT
```

```
#客户端2
#1 开启事务
start TRANSACTION     [2]
#2 查看数据
select * from user       [3]       [5]       [7]       [9]

COMMIT
```

执行到第五步会发现客户端2中事务2查询的数据发生了改变（脏读）

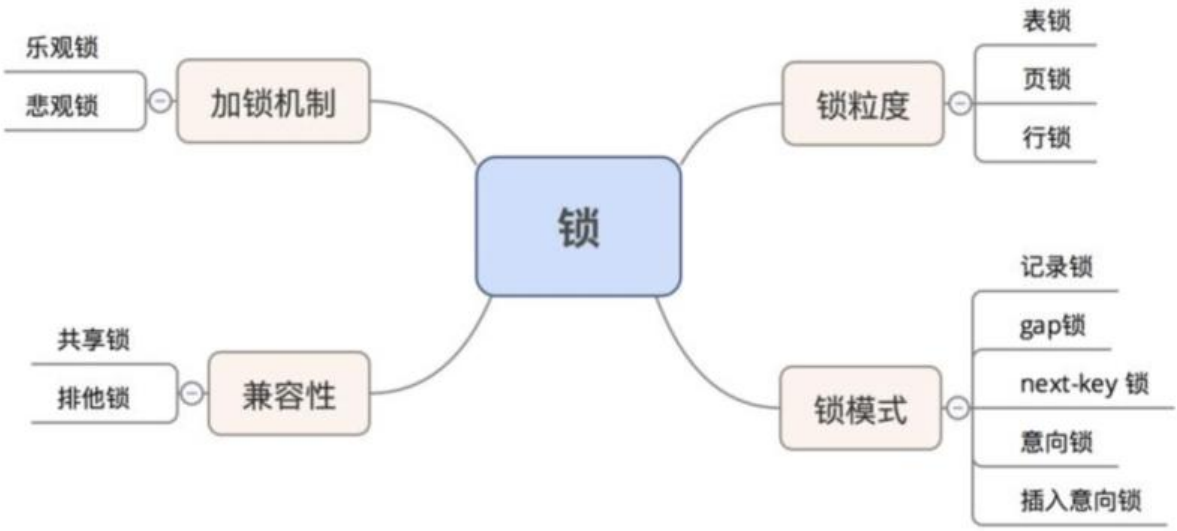
执行到第七步会发现客户端2中事务2查询的数据（id =5）不见了（不可重复读）

执行到第九步会发现客户端2中事务2查询的数据多了一条（id =7）（幻读）

注：幻读与不可重复读类似，不可重复读是读到了其他事务update/delete的结果，幻读是读到了其他事务insert的结果，~~分这么细真是觉得没啥必要~~

锁的分类

锁的种类繁多比如读锁、写锁、共享锁、互斥锁、行锁等等各种名词，根据自己的理解，简单对这些进行了分类。



加锁机制：

1. 乐观锁：先修改，保存时判断是够被更新过，应用级别（CAS:Compare-and-Swap，比较并替换）
2. 悲观锁：先获取锁，再操作修改，数据库级别

锁粒度：

表级锁：开销小，加锁快，粒度大，锁冲突概率大，并发度低，适用于读多写少的情况。

页级锁：BDB存储引擎（数据库数据按页存储）

行级锁：Innodb存储引擎，默认选项

兼容性：

● S锁：也叫做读锁、共享锁，对应于我们常用的 select * from users where id =1 lock in share mode

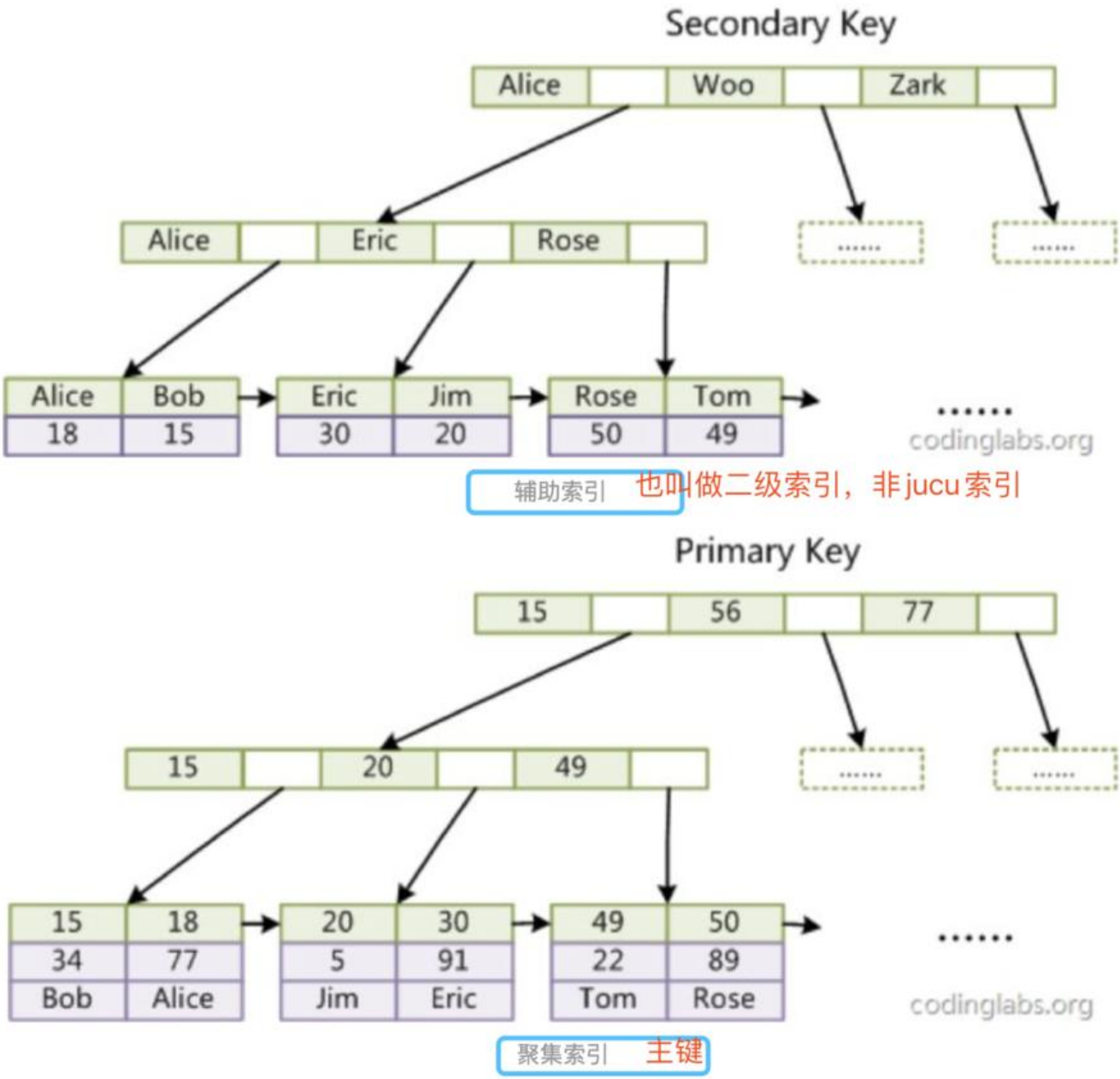
● X锁：也叫做写锁、排它锁、独占锁、互斥锁，对应对于select * from users where id =1 for update

下面这个表格是锁冲突矩阵，可以看到只有读锁和读锁之间兼容的，写锁和读锁、写锁都是冲突的。

	X	S
X	冲突	冲突
S	冲突	兼容

冲突的时候会阻塞当前会话，直到拿到锁或者超时，开启两个事务这个很好模拟。
这里要提到的一点是，S锁 和 X锁是可以是表锁，也可以是行锁。

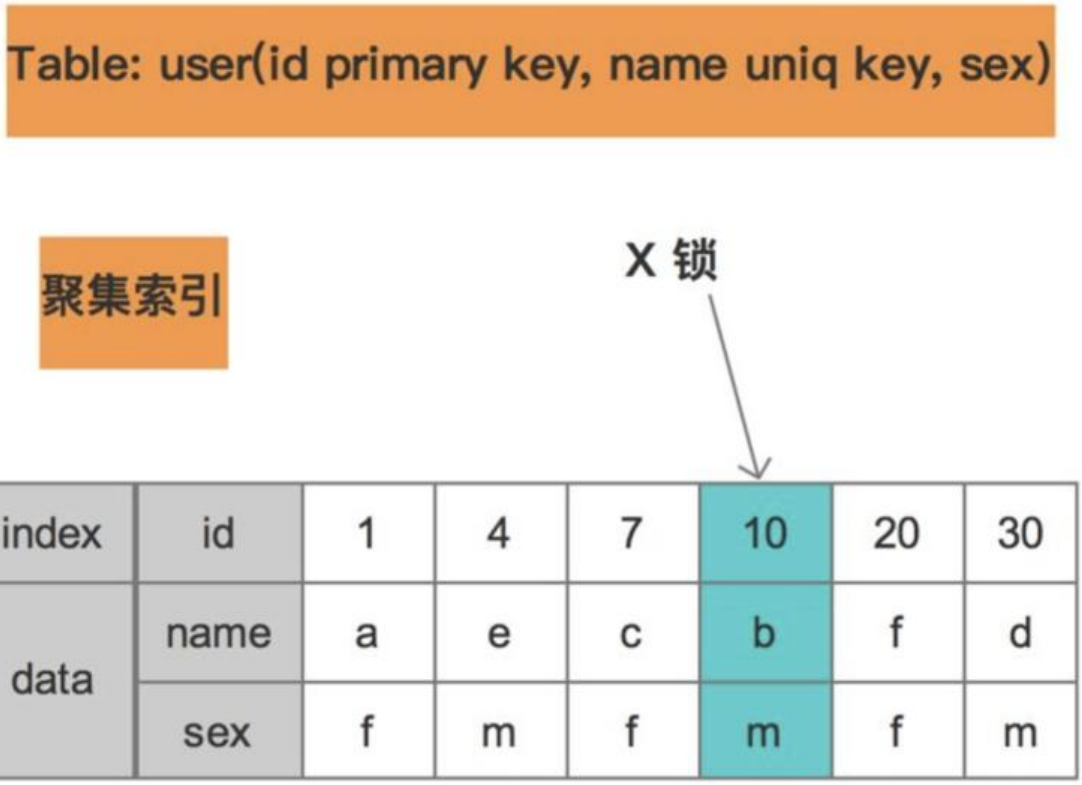
索引组织表



Innodb中的索引数据结构是 B+ 树，数据是有序排列的，从根节点到叶子节点一层层找到对应的数。普通索引，也叫做辅助索引，该（叶子leaves
节点存放的是主键值。主键上的索引叫做聚集索引，表里的每一条记录都存放在主键的节点上。当通辅助索引select 查询数据的时候，会先在辅助索引中找到对应的主键值，然后用主键值在聚集索引中到该条记录。举个例子，用name=Alice来查询的时候，会先找到对应的主键值是18 ，然后用18在下的聚集索引中找到name=Alice的记录内容是 77 和 Alice。

了解了索引数据结构的目的是为了说明，**行锁是加在索引上的。**

1.select * from user where id=10 for update



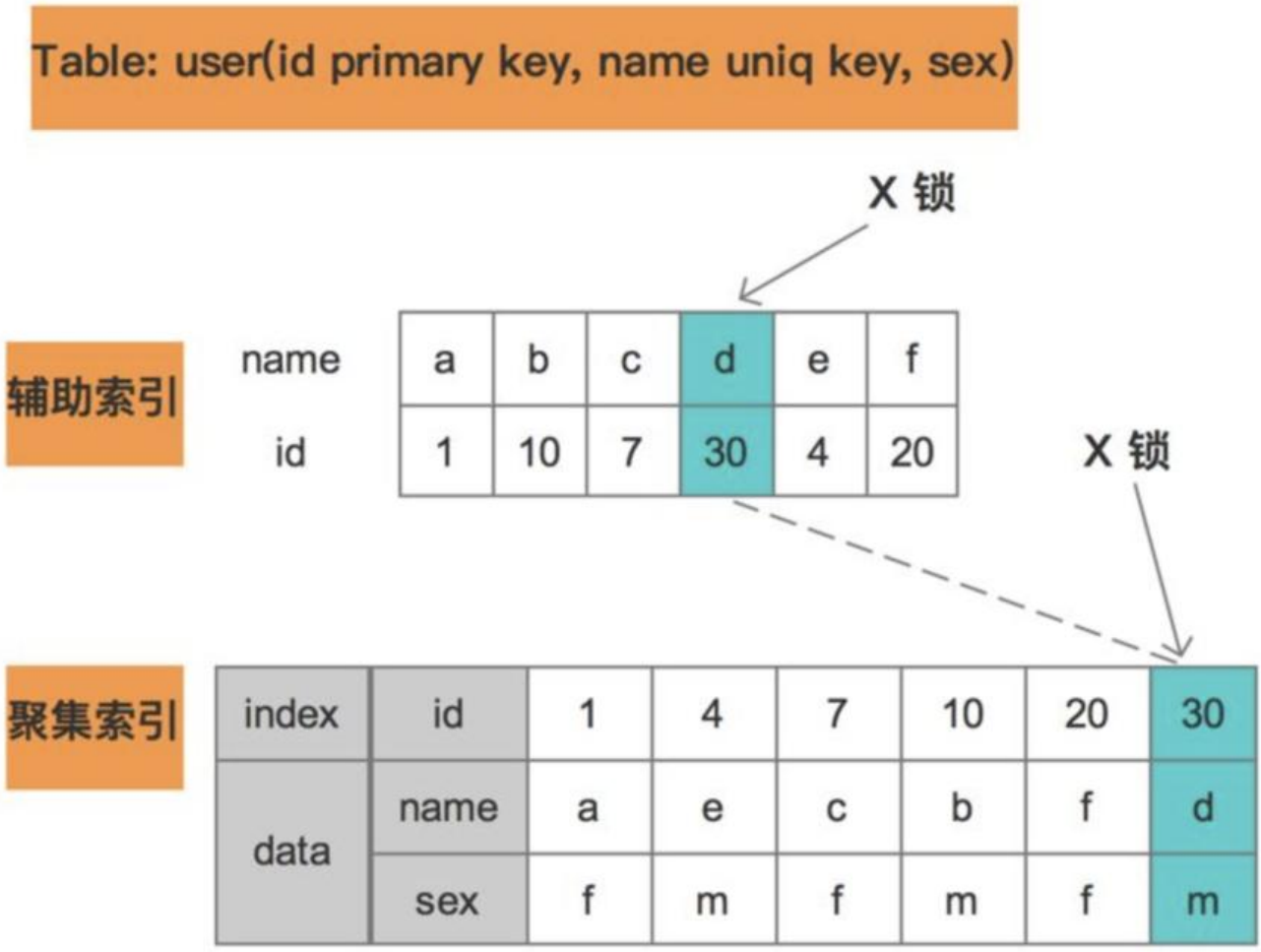
一条简单的SQL。在user表中查找id为10的记录，并用for update加X锁。

这里User表中，有3个字段， 主键id 和 另外一个字段name。下面的表格是B+树索引的简化表达。一行id是索引的节点，第二行和第三行是这行记录，包含了姓名和性别。

如图所示，通过锁住聚集索引中的节点来锁住这条记录。

聚集索引上的锁，比较好理解，锁住id=10的索引，即锁住了这条记录。

2. select * from user where name= ‘d’ for update

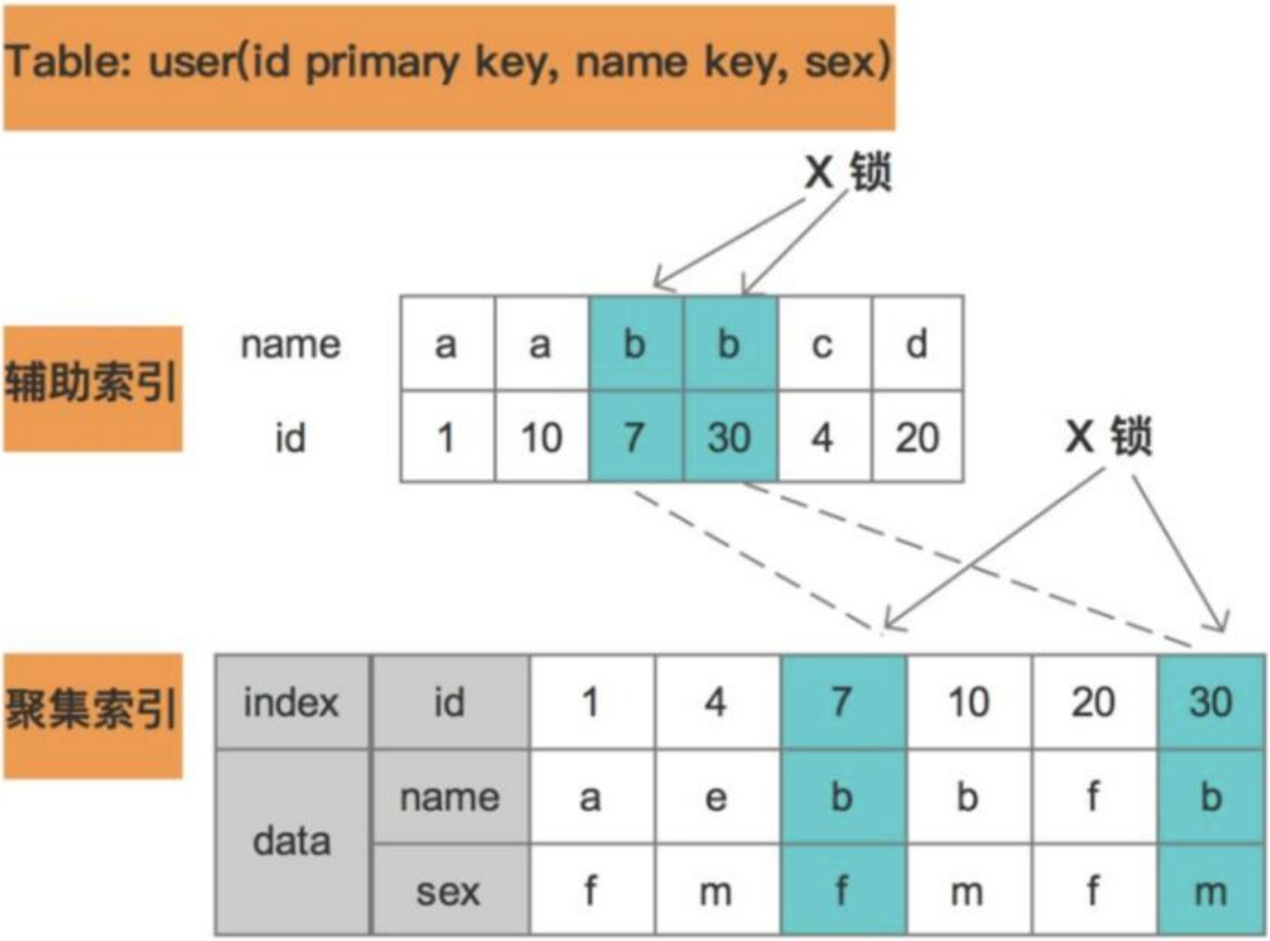


查询user表中name为d的记录，并用for update加X锁

这里的name上加了唯一索引，唯一索引本质上是辅助索引，加了唯一约束。所以会先在辅助索引上到name为d的索引记录，在辅助索引中加锁，然后查找聚集索引，锁住对应索引记录。

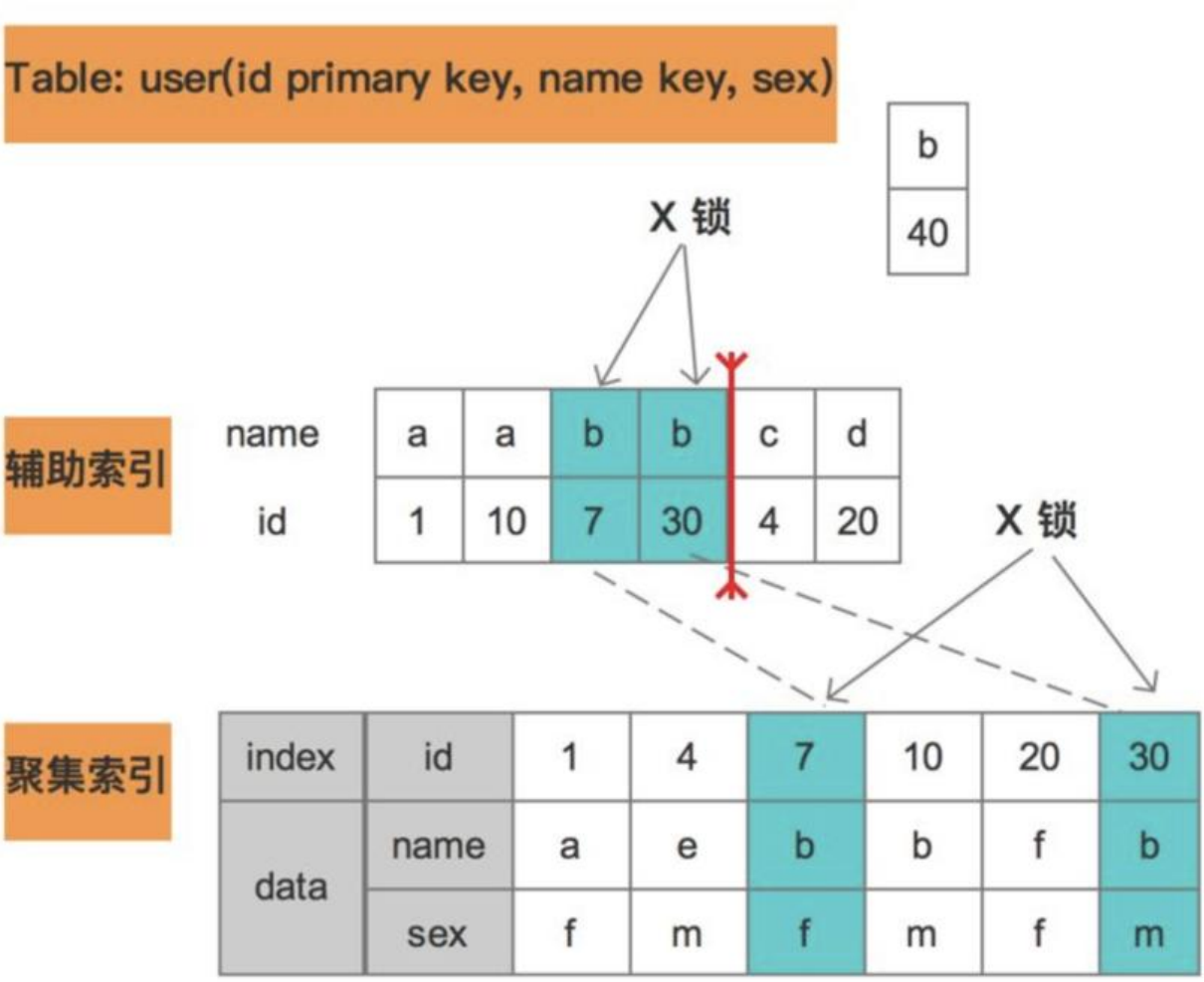
为什么聚簇索引上的记录也要加锁？试想一下，如果有并发的另外一个SQL，是直接通过主键索引id=0来更新，会先在聚集索引中请求加锁。如果只在辅助索引中加锁的话，两个并发SQL之间是互相感不到的。

3. select * from user where name= ‘b’ for update



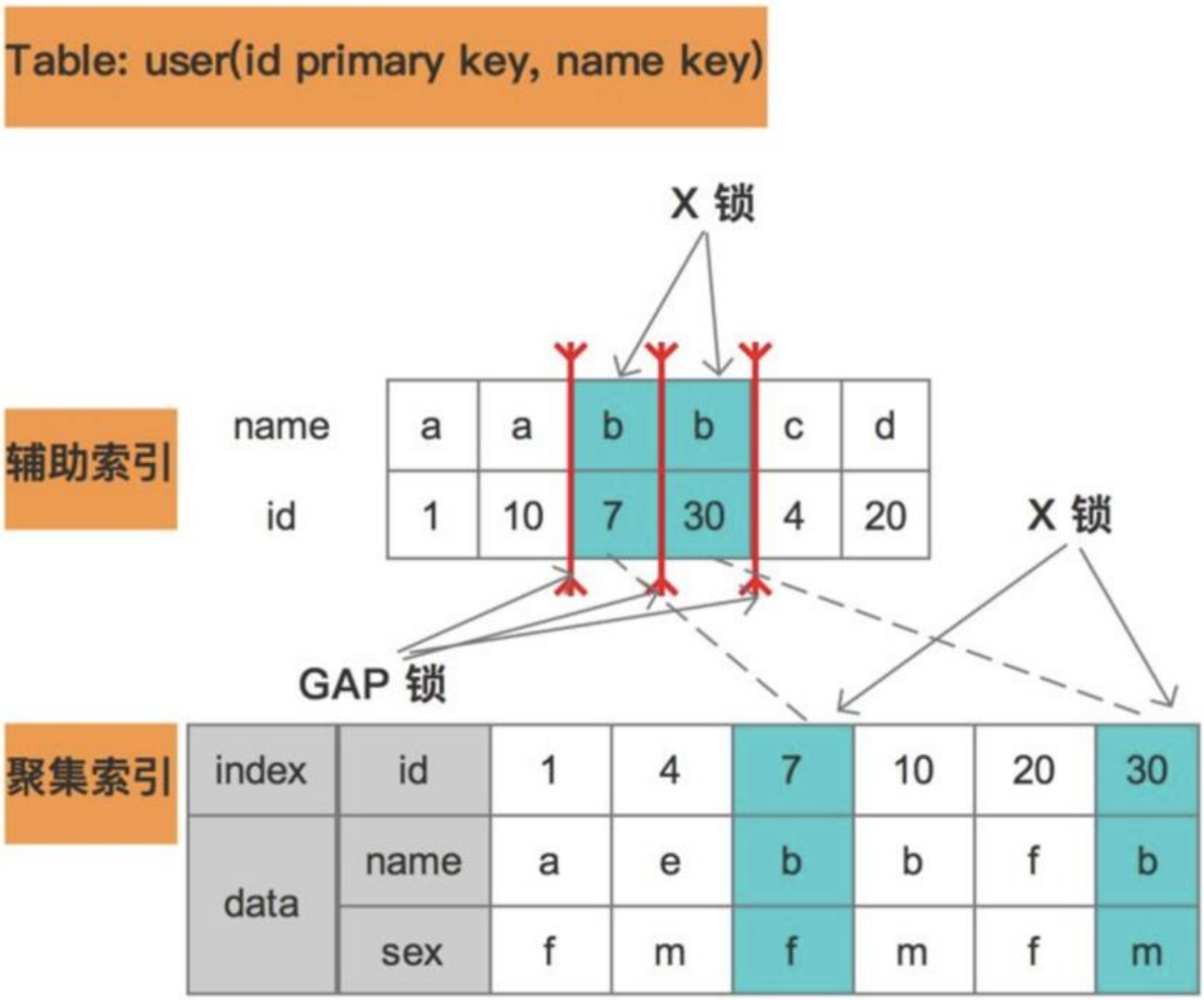
查询user表中name为b的记录，并用for update加X锁。这里name上加了普通的索引，不是唯一索。普通索引的值是可以重复的。会先在辅助索引中找到name为b的两条记录，加X锁，然后得到主键7和30，到聚集索引中加X锁。

事情并没有那么简单，如果这时有另一个事务，插入了name=b,id=40的记录，却发现是可以插入的。



位置在途中红色线条标注的间隙内，这样就会出现幻读，两次查询得到的结果是不一致的，第一次查两条数据，插入之后得到三条数据。

为了防止这种情况，出现了另一种锁，gap lcok 间隙锁。锁住的是索引的间隙。



即图中，红色线条标识的空隙。因为新插入name=b的记录，可能出现在这三个间隙内。

这张图里出现了三种锁

记录锁：单行记录上的锁

间隙锁：锁定记录之间的范围，但不包含记录本身。

Next Key Lock: 记录锁+ 间隙锁，锁定一个范围，包含记录本身。

4. 意向锁(Intention Locks)

InnoDB为了支持多粒度(表锁与行锁)的锁并存，引入意向锁。意向锁是表级锁，

IS: 意向共享锁

IX: 意向排他锁

事务在请求某一行的S锁和X锁前，需要先获得对应表的IS、IX锁。

意向锁产生的主要目的是为了处理行锁和表锁之间的冲突，用于表明“某个事务正在某一行上持有了，或者准备去持有锁”。比如，表中的某一行上加了X锁，就不能对这张表加X锁。

如果不在表上加意向锁，对表加锁的时候，都要去检查表中的某一行上是否加有行锁，多麻烦。

请求\持有	IS	IX	S	X
IS	兼容	兼容	兼容	冲突
IX	兼容	兼容	冲突	冲突
S	兼容	冲突	兼容	冲突
X	冲突	冲突	冲突	冲突

意向锁的兼容性矩阵

5. 插入意向锁（Insert Intention Lock）

Gap Lock中存在一种插入意向锁，在insert操作时产生。

有两个作用：

- 和next-key互斥，阻塞next-key 锁，防止插入数据，这样就不会幻读。
- 插入意向锁互相是兼容的，允许相同间隙、不同数据的并发插入

加锁分析

后面会有多个SQL语句，先说明一下表结构

```
CREATE TABLE `user` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `id_no` varchar(255) DEFAULT NULL COMMENT '身份证号',
  `name` varchar(255) DEFAULT NULL COMMENT '姓名',
  `mobile` varchar(255) DEFAULT NULL COMMENT '手机号',
  `age` int(11) DEFAULT NULL COMMENT '年龄',
  `address` varchar(255) DEFAULT NULL COMMENT '地址',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uniq_id_no` (`id_no`),
  KEY `idx_name` (`name`)
) ENGINE=InnoDB AUTO_INCREMENT=10002 DEFAULT CHARSET=utf8 COMMENT='用户表';
```

这里有一个user表，5个字段，其中id是主键，id_no是身份证号，加了唯一索引，name是用户姓名可以重复的，加了普通索引，手机号、年龄、地址都没有索引。

1. 普通select

```
select  * from user where id =1;

begin;
select  * from user where id =1;
commit:
```

普通的select 语句是不加锁的。select包裹在事务中，同样也是不加锁的。where后面的条件不管多，普通的select是不加锁的。

2. 显式加锁

```
select  * from user where id =1 lock in share mode;

select  * from user where id =1 for update;
```

显式指出要加什么样的锁。上面一个加的是共享锁，下面的是互斥锁。

这里需要强调的一点，需要明确在事务中是用这些锁，不在事务中是没有意义的。

3. 隐式加锁

```
update user set address '北京' where id=1;
delete from user where id=1;
```

update和delete也会对查询出的记录加X锁，隐式加互斥锁。加锁类型和for update 类似。

4. 按索引类型

```
select  * from user where id =1 for update;

select  * from user where id_no ='a22' for update;

select  * from user where name ='王二' for update;

select  * from user where address ='杭州' for update;
```

四条SQL，区别在于where条件的过滤列，分别是主键、唯一索引、普通索引、无索引。

主键	聚集索引 + X锁
唯一索引	辅助索引 + X锁 聚集索引 + X锁
普通索引	辅助索引 + next-key锁 聚集索引 + X锁
无索引	聚集索引|全表+next-key

主键：之前提到过索引组织表，这里会在聚集索引上对查询出的记录，加X锁

唯一索引：会在辅助索引上，把在对应的id_no=a22的索引加X锁，因为是唯一的，所以不是next-ke锁。然后在主键上，也会在这条记录上加X锁。

普通索引：因为不是唯一的，会在辅助索引上，把对应的id_no=a22的索引加next-key锁。然后在主加X锁。

无索引：首先，是不推荐这种写法，没有索引的话，因为会全表扫描，数据量大的话查询会很慢。这讨论的是，这种情况下，会加什么锁？ 答案： 首先，聚簇索引上的所有记录，都被加上了X锁。其次聚簇索引每条记录间的间隙(GAP)，也同时被加上了GAP锁。在这种情况下，这个表上，除了不加锁快照度，其他任何加锁的并发SQL，均不能执行，不能更新，不能删除，不能插入，全表被锁死。这一个很恐怖的事情，请注意。

5. 记录不存在的情况

前面几个例子中，都是可以查到结果的。如果对应记录不存在会怎样？答案是锁住间隙，不允许插入mysql要保证没有其他人可以插入，所以锁住间隙。

6. 普通 insert 语句

在插入之前，会先在插入记录所在的间隙加上一个插入意向锁。

insert会对插入成功的行加上排它锁，这个排它锁是个记录锁，而非next-key锁（当然更不是gap锁），不会阻止其他并发的事务往这条记录之前插入 。

分析死锁

先说一下死锁的定义，**死锁是指两个或两个以上的事务在执行过程中，因争夺资源而造成的一种互相待的现象。**这个定义适用于数据库，有几个重点，两个或两个以上的事务，一个事务是不会出现死锁。争夺的资源一般都是表或者记录。

出现死锁了会怎样，正常情况下，mysql会检查出死锁，并回滚某一个事务，让另一个事务正常运行。

Mysql 会回滚副作用小的事务，判定的标准是执行的时间以及影响的范围。

1.如何知道系统有没有发生过死锁，如何去查看发生过的锁

show status like 'innodb_row_lock%'

Variable_name	Value
Innodb_row_lock_current_waits	0
Innodb_row_lock_time	28767
Innodb_row_lock_time_avg	10
Innodb_row_lock_time_max	493
Innodb_row_lock_waits	2718

Innodb_row_lock_current_waits：当前正在等待锁的数量；

Innodb_row_lock_time ： 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。

平均时间和锁等待次数比较大的话，说明可能会存在锁争用情况

查看innodb存储引擎的运行状态

show engine innodb status

```
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
---------------------+
| InnoDB |      |
=====================================
2019-11-07 08:23:12 0x7fb25c0f9700 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 60 seconds
-----------------
BACKGROUND THREAD
-----------------
srv_master_thread loops: 18993 srv_active, 0 srv_shutdown, 247775 srv_idle
srv_master_thread log flush and writes: 0
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 1196
OS WAIT ARRAY INFO: signal count 64765
RW-shared spins 57070, rounds 57253, OS waits 148
RW-excl spins 34288, rounds 51281, OS waits 86
RW-sx spins 25, rounds 223, OS waits 2
Spin rounds per wait: 1.00 RW-shared, 1.50 RW-excl, 8.92 RW-sx
------------------------
LATEST FOREIGN KEY ERROR
------------------------
2019-11-05 02:46:46 0x7fb2570f6700 Transaction:
TRANSACTION 161185, ACTIVE 0 sec inserting
mysql tables in use 1, locked 1
13 lock struct(s), heap size 1136, 5 row lock(s), undo log entries 5
MySQL thread id 162366, OS thread handle 140403941533440, query id 1285106 10.5.6.16 senseguard update
INSERT INTO info_r_task_library_timezone  ( task_id,library_id,time_zone_id,remark )  VALUES   ( 4,2,10,nu
Foreign key constraint fails for table `senseguard`.`info_r_task_library_timezone`:
,
  CONSTRAINT `info_r_task_library_timezone_ibfk_3` FOREIGN KEY (`time_zone_id`) REFERENCES `info_time_zone
Trying to add in child table, in index info_r_task_library_timezone_ibfk_time_zone_id tuple:
DATA TUPLE: 2 fields;
 0: len 8; hex 800000000000000a; asc         ;;                    下面还有很多
 1: len 8; hex 8000000000000001; asc         ;;
```

通过这个命令显示的内容比较多，其中有一项lasted detected deadlock 显示最近发生的死锁。以及了什么锁。

查看已开启事务的状态

select * from information_schema.innodb_trx;

trx_id：innodb存储引擎内部事务唯一的事务id。

trx_state：当前事务的状态。

trx_started：事务开始的时间。

trx_requested_lock_id：等待事务的锁id，如trx_state的状态为LOCK WAIT，那么该值代表当前事之前占用锁资源的id，如果trx_state不是LOCK WAIT的话，这个值为null。

trx_wait_started：事务等待开始的时间。

trx_weight：事务的权重，反映了一个事务修改和锁住的行数。在innodb的存储引擎中，当发生死需要回滚时，innodb存储引擎会选择该值最小的事务进行回滚。

trx_mysql_thread_id：正在运行的mysql中的线程id，show full processlist显示的记录中的thread_i。

trx_query：事务运行的sql语句

完结~