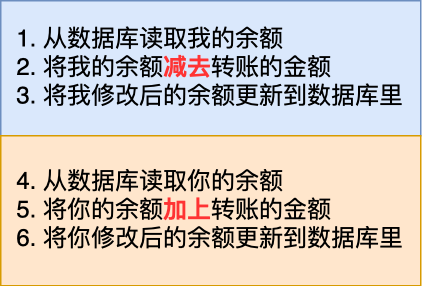
**事务隔离级别是怎么实现的？**



转账的过程涉及到了两次修改数据库的操作。执行第三步骤之后，服务器忽然掉电了，就会导致我的账户扣了 100 万，但是钱并没有到你的账户上，也就是说**这 100 万消失了！**

要解决这个问题，就要**保证转账业务里的所有数据库的操作是不可分割的，要么全部执行成功 ，要么全部失败，不允许出现中间状态的数据**。

数据库中的「**事务（Transaction）**」就能达到这样的效果：在转账操作前先开启事务，等所有数据库操作执行完成后，才提交事务，**对于已经提交的事务来说，该事务对数据库所做的修改将永久生效**，如果**中途发生发生中断或错误，那么该事务期间对数据库所做的修改将会被回滚到没执行该事务之前的状态**。

## 事务的特性

InnoDB 引擎支持事务， MyISAM 引擎不支持事务

事务的 4 个特性

* **原子性（Atomicity）**：**一个事务中的所有操作，要么全部完成，要么全部不完成**，不会结束在中间某个环节，而且事务在执行过程中**发生错误，会被回滚到事务开始前的状态**。
* **一致性（Consistency）**：是指事务操作前和操作后，**数据满足完整性约束，数据库保持一致性状态**。比如转账前后两个账户的余额总数保持一致。
* **隔离性（Isolation）**：数据库允许多个**并发事务同时对其数据进行读写和修改**的能力。多个事务同时使用相同的数据时，对其他并发事务是隔离的，不会相互干扰，每个事务都有一个完整的数据空间。
* **持久性（Durability）**：事务处理结束后，**对数据的修改就是永久的**，即便系统故障也不会丢失。

InnoDB 引擎保证事务的四个特性的方法：

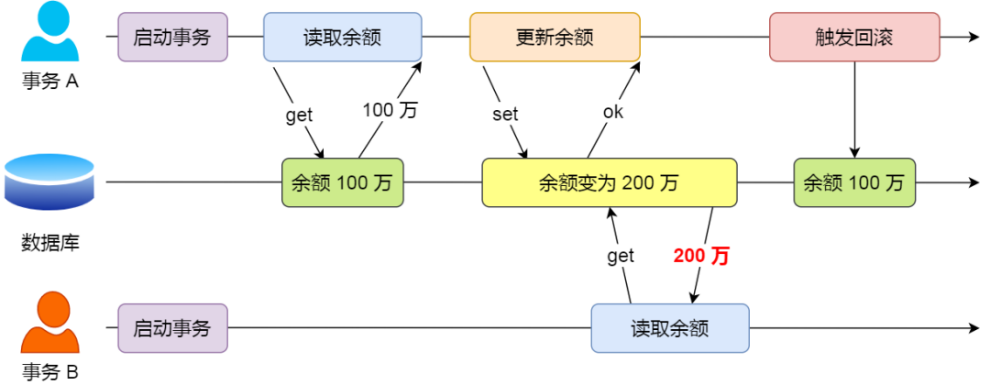
* 持久性：通过 **redo log （重做日志）**来保证的；
* 原子性：通过 **undo log（回滚日志）** 来保证的；
* 隔离性：通过 **MVCC（多版本并发控制）** 或**锁机制**来保证的；
* 一致性：通过持久性+原子性+隔离性来保证；

## 并行事务会引发什么问题？

**在同时处理多个事务的时候，就可能出现脏读（dirty read）、不可重复读（non-repeatable read）、幻读（phantom read）的问题**。

### 脏读

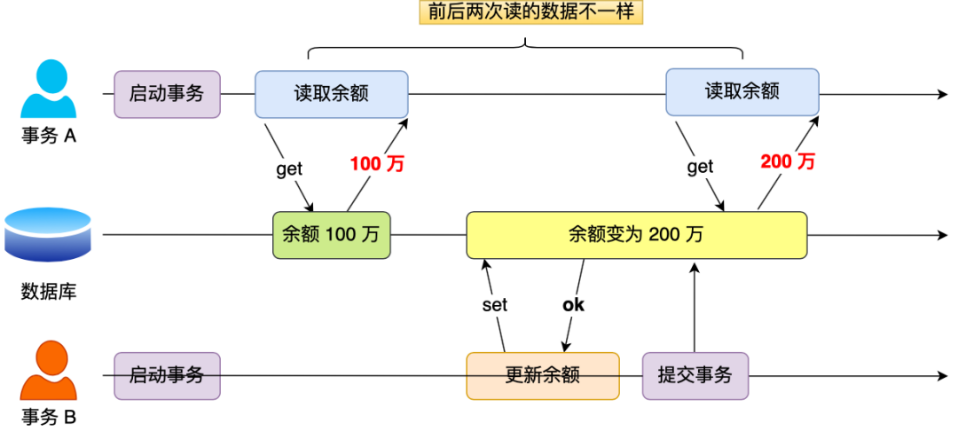
指**一个事务「读到」了另一个「未提交事务修改过的数据」**



事务 A还未提交事务，随时可能发生回滚操作，那么**事务 B得到的数据就是过期的数据**

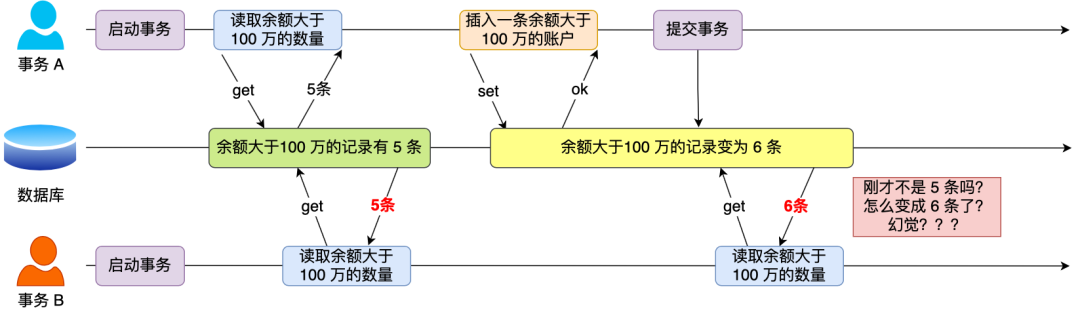
### 不可重复读

指**在一个事务内多次读取同一个数据，出现前后两次读到的数据不一样的情况**



### 幻读

指**在一个事务内多次查询某个符合查询条件的「记录数量」，出现前后两次查询到的记录数量不一样的情况**



## 事务的隔离级别有哪些？

* 脏读：读到其他事务未提交的数据；
* 不可重复读：前后读取的数据不一致；
* 幻读：前后读取的记录数量不一致。

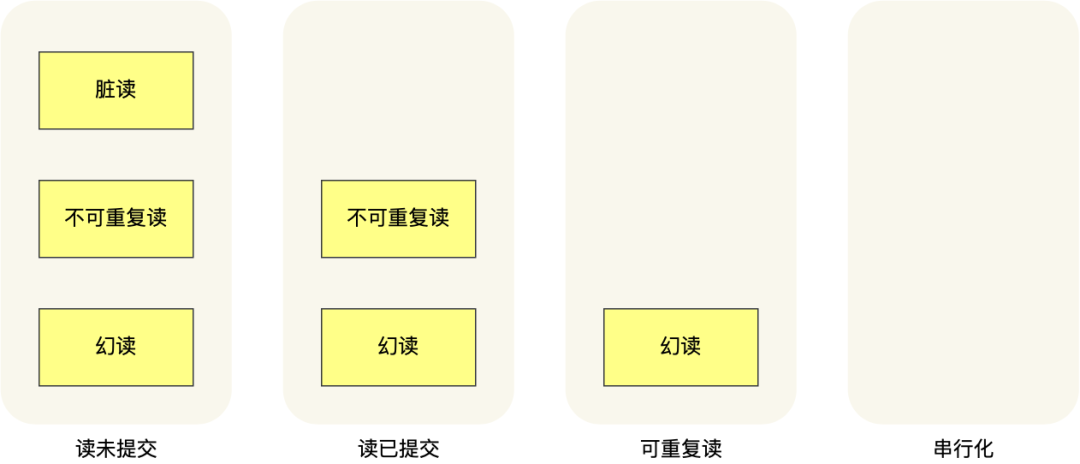
严重性排序：**脏读>不可重复读>幻读**

SQL 标准提出了四种隔离级别来规避这些现象，**隔离级别越高，性能效率就越低**：

* **读未提交（*read uncommitted*）**：指一个事务还没提交时，它做的变更就能被其他事务看到；
* **读提交（*read committed*）**：指一个事务提交之后，它做的变更才能被其他事务看到；
* **可重复读（*repeatable read*）**：指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，**MySQL InnoDB 引擎的默认隔离级别**；
* **串行化（*serializable* ）**：会对记录加上**读写锁**，在多个事务对这条记录进行读写操作时，如果**发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成才能继续执行**；

隔离水平高低排序：**串行化>可重复读>读提交>读未提交**

不同的隔离级别，并发事务时可能发生的现象：



不同的数据库厂商对 SQL 标准中规定的 4 种隔离级别的支持不一样。**MySQL 虽然支持 4 种隔离级别，但是与SQL 标准中规定的各级隔离级别允许发生的现象却有些出入。MySQL InnoDB 引擎的默认隔离级别虽然是「可重复读」，但是它很大程度上避免幻读现象。**MySQL **并不会使用「串行化」隔离级别**来避免幻读现象的发生，因为使用「串行化」隔离级别会影响性能。

四种隔离级别的实现方法：

* 读未提交：直接读取最新的数据；
* 串行化：加**读写锁**来避免并行访问；
* 读提交和可重复读：通过 **Read View 来实现的。「读提交」隔离级别是在「每个语句执行前」都会重新生成一个 Read View，而「可重复读」隔离级别是「启动事务时」生成一个 Read View，然后整个事务期间都在用这个 Read View**。

MySQL 有两种开启事务的命令：

* **begin/start transaction 命令**：执行这个命令后，执行了第一条 select 语句，事务才真正启动；
* **start transaction with consistent snapshot 命令**：马上启动事务。

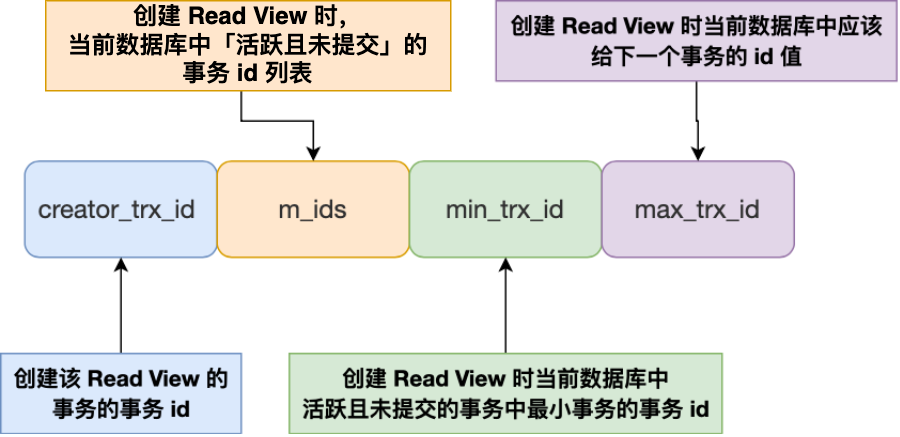
## Read View 在 MVCC 里如何工作的？

**通过「版本链」来控制并发事务访问同一个记录时的行为就叫 MVCC（多版本并发控制）**

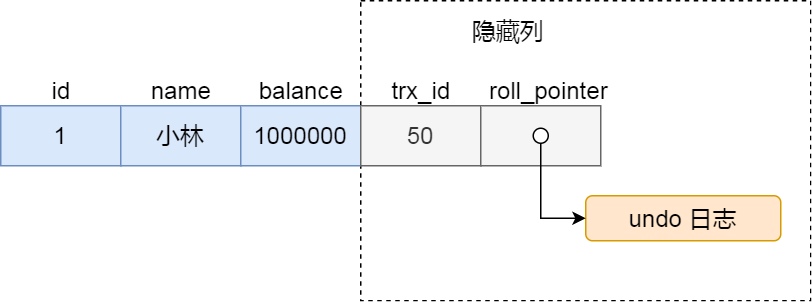
ReadView可以被理解为一个“快照”，它**定义了事务在某一时刻能看到的数据版本范围**。

Read View 有四个重要的字段：

* creator\_trx\_id ：指的是**创建该 Read View 的事务的事务 id**。
* m\_ids ：指的是在创建 Read View 时，当前数据库中**活跃事务（启动了但还没提交的事务）**的**事务 id 列表**。
* min\_trx\_id ：指的是在创建 Read View 时，当前数据库中**活跃事务中事务 id 最小的事务**，即 m\_ids 中的最小值。
* max\_trx\_id ：这个并不是 m\_ids 的最大值，而是创建 Read View 时**当前数据库中应该给下一个事务的 id 值**，也就是**全局事务中最大的事务 id 值 + 1**；

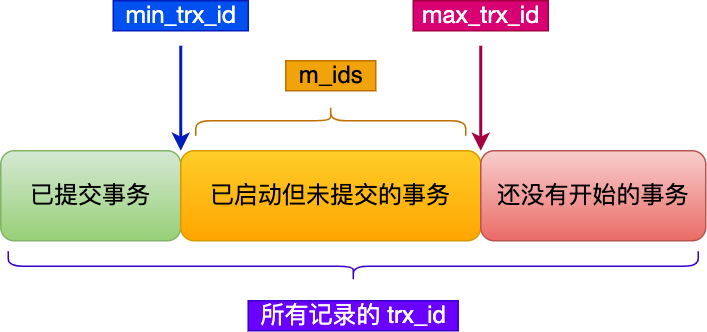


聚簇索引记录中与MVCC相关的两个隐藏列（还有一个ROW\_ID和MVCC没什么关系）：



* trx\_id：当一个事务**对某条聚簇索引记录进行改动**时，就会**把该事务的事务 id 记录在 trx\_id 隐藏列里**；
* roll\_pointer：每次对某条聚簇索引记录进行改动时，都会**把旧版本的记录写入到 undo 日志中**，然后**这个隐藏列是个指针，指向每一个旧版本记录**，于是就可以通过它找到修改前的记录。

在创建 Read View 后，可以将记录中的 trx\_id 划分这三种情况：



注意这个图并不是和计算机网络滑动窗口一样连续的事务

一个事务去访问记录时，有这几种情况：

* 如果**记录的trx\_id值就是当前事务**，**可见**。
* 如果**记录的 trx\_id 值小于 Read View 中的 min\_trx\_id 值**，表示这个版本的记录是在**创建 Read View 前已经提交的事务生成**的，所以该版本的记录对当前事务**可见**。
* 如果**记录的 trx\_id 值大于等于 Read View 中的 max\_trx\_id 值**，表示这个版本的记录是在**创建 Read View 后才启动的事务生成**的，所以该版本的记录对当前事务**不可见**。
* 如果**记录的 trx\_id 值在 Read View 的 min\_trx\_id 和 max\_trx\_id 之间**，需要判断 trx\_id 是否在 m\_ids 列表中：
  + 如果**记录的 trx\_id 在 m\_ids 列表中**，表示生成该版本记录的活跃事务**依然活跃着（还没提交事务）**，所以该版本的记录对当前事务**不可见**。
  + 如果**记录的 trx\_id 不在 m\_ids列表中**，表示生成该版本记录的活跃事务**已经被提交**，所以该版本的记录对当前事务**可见**。

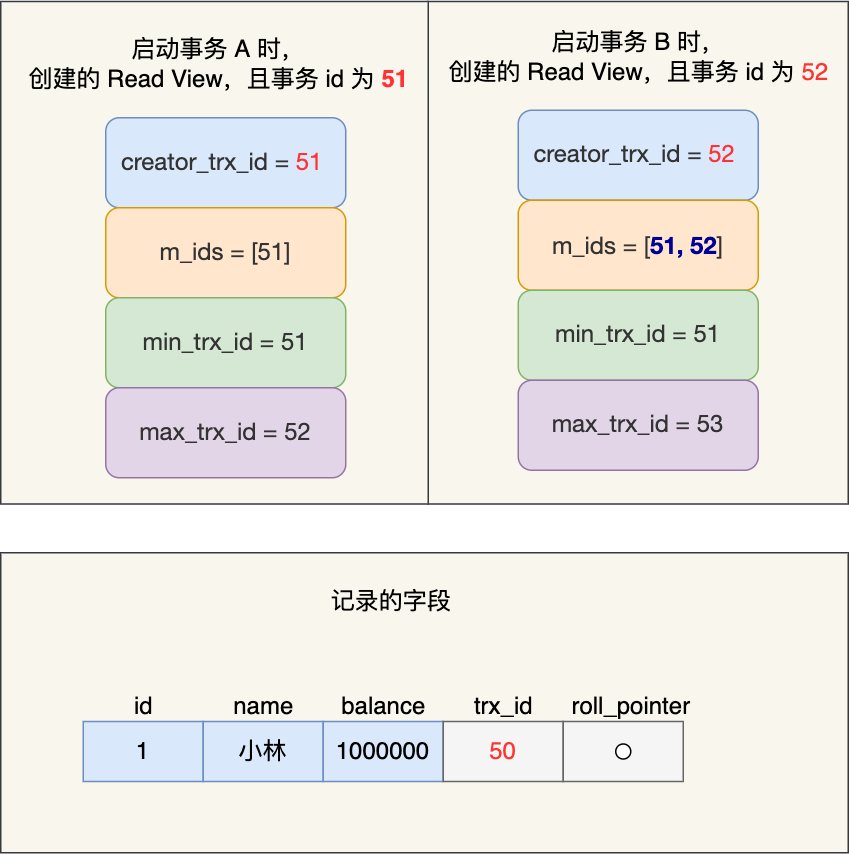
## 可重复读是如何工作的？

**可重复读隔离级别是启动事务时生成一个 Read View，然后整个事务期间都在用这个 Read View**。

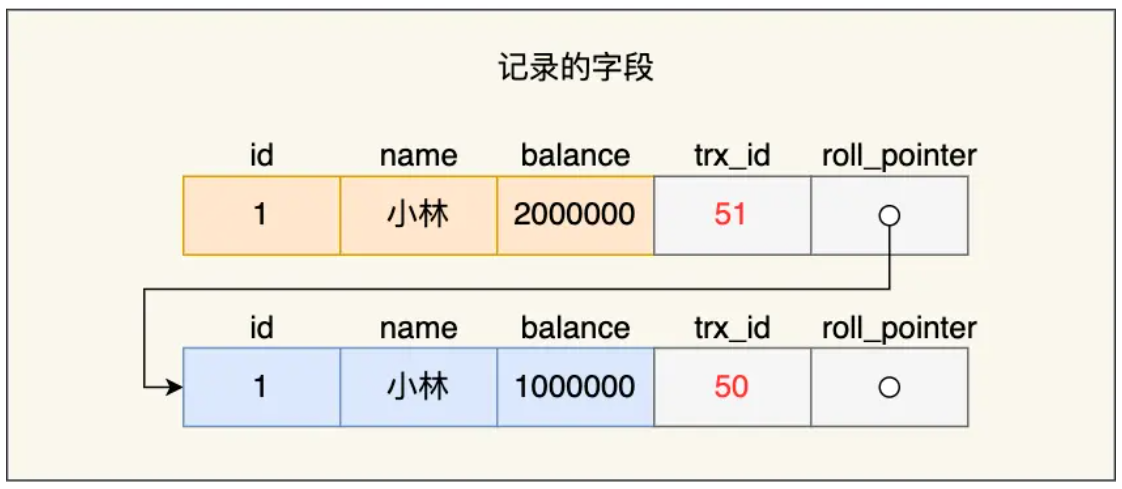
举例：

* 事务 B 读取小林的账户余额记录，读到余额是 100 万；
* 事务 A 将小林的账户余额记录修改成 200 万，并没有提交事务；
* 事务 B 读取小林的账户余额记录，读到余额还是 100 万；
* 事务 A 提交事务；
* 事务 B 读取小林的账户余额记录，读到余额依然还是 100 万；

① 事务 B 第一次读小林的账户余额记录，此时**发现 trx\_id 为 50，比事务 B 的 Read View 中的 min\_trx\_id 值（51）还小，这意味着修改这条记录的事务早就在事务 B 启动前提交过了，所以该版本的记录对事务 B 可见的**



② 事务 A 通过 update 语句将这条记录修改了（还未提交事务），将余额改成 200 万，这时 MySQL 会记录相应的 undo log，并以链表的方式串联起来，形成**版本链：**



③ 事务 B 第二次去读取该记录，**发现这条记录的 trx\_id 值为 51，在事务 B 的 Read View 的 min\_trx\_id 和 max\_trx\_id 之间，则需要判断 trx\_id 值是否在 m\_ids 范围内，判断的结果是在的，那么说明这条记录是被还未提交的事务修改的，这时事务 B 并不会读取这个版本的记录。而是沿着 undo log 链条往下找旧版本的记录，直到找到 trx\_id 「小于」事务 B 的 Read View 中的 min\_trx\_id 值的第一条记录（？好像也要和上面一样判断m\_ids）**，所以事务 B 能读取到的是 trx\_id 为 50 的记录，也就是小林余额是 100 万的这条记录。

④ 事务 A提交事务

⑤ **事务 B 第三次读取记录时，还是基于启动事务时创建的 Read View 来判断当前版本的记录是否可见。仍然和③一样，读到的记录都是小林余额是 100 万的这条记录**。

## 读提交是如何工作的？

**读提交隔离级别是在每次读取数据时，都会生成一个新的 Read View**。

其余步骤与可重复读一致，只有⑤中，由于每次读取会生成一个新的Read View，此时的min\_trx\_id变为52，而该记录中的trx\_id为51，因此变为**可见**，读取到200万

# MySQL 可重复读隔离级别，完全解决幻读了吗？

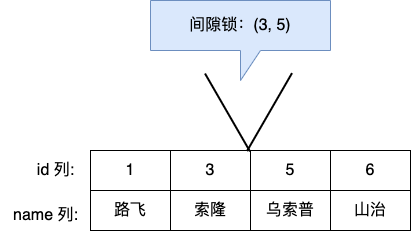
## 快照读是如何避免幻读的？

针对**快照读**（**普通 select 语句**），是**通过 MVCC 方式解决了幻读**，因为可重复读隔离级别下，事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，即使中途有其他事务插入了一条数据，是查询不出来这条数据的，所以就很好了避免幻读问题。

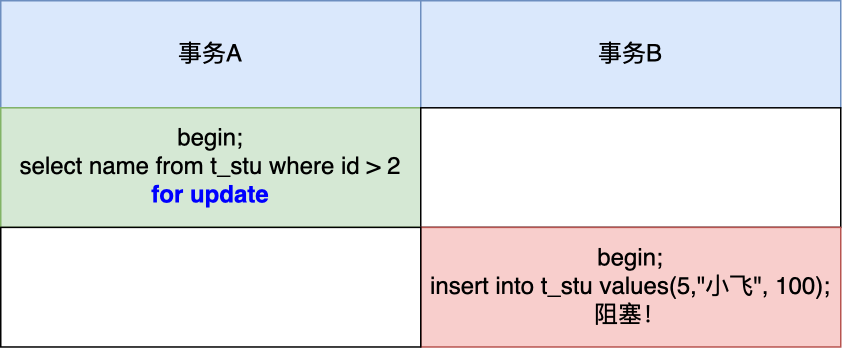
## 当前读是如何避免幻读的？

MySQL 里除了普通查询是快照读，其他都是**当前读**，比如 update、insert、delete，这些语句执行前都会**查询最新版本的数据，然后再做进一步的操作**。

InnoDB在**「可重复读」隔离级别**针对**当前读**（**select ... for update 等语句**），是**通过 next-key lock（记录锁+间隙锁）方式解决了幻读**，因为当执行 select ... for update 语句的时候，会加上 next-key lock，如果有其他事务在 next-key lock 锁范围内插入了一条记录，那么这个插入语句就会被阻塞，无法成功插入，所以就很好了避免幻读问题。



举例：



事务 A 执行这条锁定读语句后，就在对表中的记录加上 **id 范围为 (2, +∞] 的 next-key lock**。然后，事务 B 在执行插入语句的时候，判断到插入的位置被事务 A 加了 next-key lock，于是事物 B 会生成一个**插入意向锁**，同时**进入等待状态，直到事务 A 提交了事务**。这就避免了由于事务 B 插入新记录而导致事务 A 发生幻读的现象。

## 幻读被完全解决了吗？

**没有能完全解决幻读**

主要出现在**先快照读，再当前读，中间其他事务插入数据并提交**的场景中。

### 第一个发生幻读现象的场景

举例：



① 事务 A 执行查询 id = 5 的记录（**注意是普通读不是当前读，当前读就算没有这个记录，也会创建一个[5,+∞）的next-key lock**），此时表中没有该记录，查询不出来（普通读不会加上next-key lock）。

② 然后事务 B 插入一条 id = 5 的记录，并且提交了事务。

③ **事务 A 更新 id = 5 这条记录，此时这条新记录的 trx\_id 隐藏列的值就变成了事务 A 的事务 id，使得普通select（快照读）可以进行**

④ **事务A再次查询 id = 5 的记录，就能看到事务 B 插入的纪录了，发生了幻读**。

### 第二个发生幻读现象的场景

① 事务 A 先执行「**快照读**语句」：select \* from t\_test where id > 100 得到了 3 条记录。

② 事务 B 往插入一个 id= 200 的记录并提交；

③ 事务 A 再执行「**当前读**语句」 select \* from t\_test where id > 100 for update 就会得到 4 条记录，此时也发生了幻读现象。

避免幻读方法：**尽量在开启事务之后，马上执行 select ... for update 这类当前读的语句**，因为它会对记录加 next-key lock，从而避免其他事务插入一条新记录。