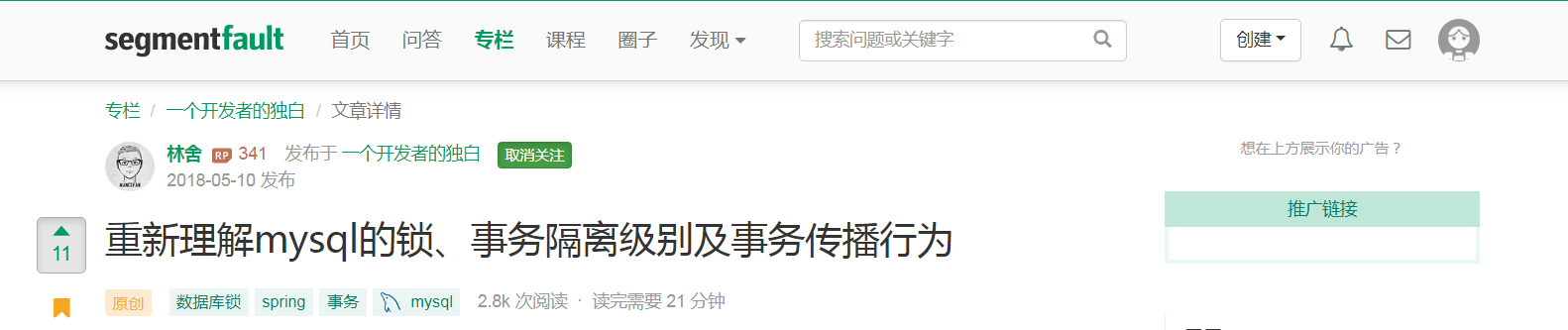
# 重新理解mysql的锁、事务隔离级别及事务传播行为

原文链接：<https://segmentfault.com/a/1190000014811125?utm_source=tag-newest#articleHeader5>

**Segmentfault 思否**



数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。  
ACID，是指在可靠数据库管理系统（DBMS）中，事务(Transaction)所应该具有的四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

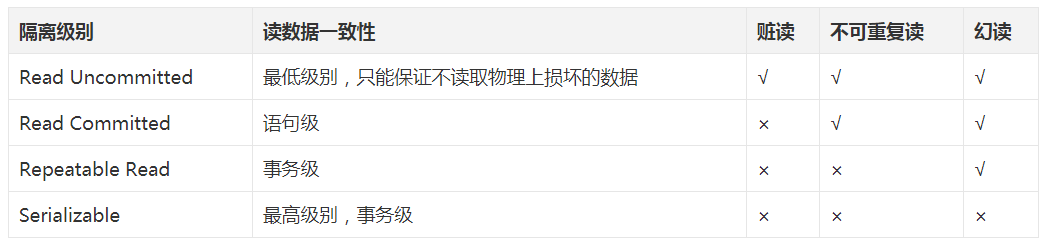
* **原子性**  
  原子性是指事务是一个不可再分割的工作单位，事务中的操作要么都发生，要么都不发生。  
  如，A向B转钱，在事务中的扣款和加款两条语句，要么都执行，要么都不执行。
* **一致性**  
  一致性是指事务使得系统从一个一致的状态转换到另一个一致状态。  
  如，A和B存款总额为1000，A向B转钱，无论失败，最终A和B的存款总额依然为1000.
* **隔离性**  
  多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。  
  数据库多个事务之间操作可能出现的问题以及事务隔离级别是这篇文章介绍的重点。
* **持久性**  
  持久性，意味着在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。  
  即使出现了任何事故比如断电等，事务一旦提交，则持久化保存在数据库中。

## 事务的并发问题

* **赃读**（Dirty Read）  
  一个事务读取到了另外一个事务没有提交的数据  
  事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据
* **不可重复读**（Nonrepeatable Read）  
  在同一事务中，两次读取同一数据，得到内容不同  
  事务A多次读取同一数据，事务B在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果不一致
* **幻读**（Phantom Read）  
  同一事务中，用同样的操作读取两次，得到的记录数不相同  
  系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样

## MySql的四种隔离级别

* **Read Uncommitted**（读取未提交内容）  
  在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。  
  读取未提交的数据，则会发生**赃读**
* **Read Committed**（读取提交内容）  
  一个事务只能看见已经提交事务所做的改变。这是大多数数据库系统的默认隔离级别，但非MySql  
  一个事务多次读取的过程中，另一个事务可能对同一条数据做修改并提交，导致前一个事务多次读取到的数据不一致，则会发生**不可重复读**
* **Repeatable Read**（可重读）  
  它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。这是MySql的默认隔离级别  
  但，此级别依然会发生**幻读**
* **Serializable**（可串行化）  
  它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题



低级别的隔离一般支持更高的并发处理，并拥有更低的系统开销。高级别的隔离可靠性较高，但系统开销较大。

### 隔离级别测试

创建数据库

**CREATE** **DATABASE** **IF** **NOT** **EXISTS** txdemo **DEFAULT** **CHARSET** utf8 **COLLATE** utf8\_general\_ci;

创建测试表

**CREATE** **TABLE** `user` (

`id` BIGINT **NOT** NULL AUTO\_INCREMENT,

`name` VARCHAR(32) **NOT** NULL **DEFAULT** '',

`age` INT(16) **NOT** NULL **DEFAULT** '30',

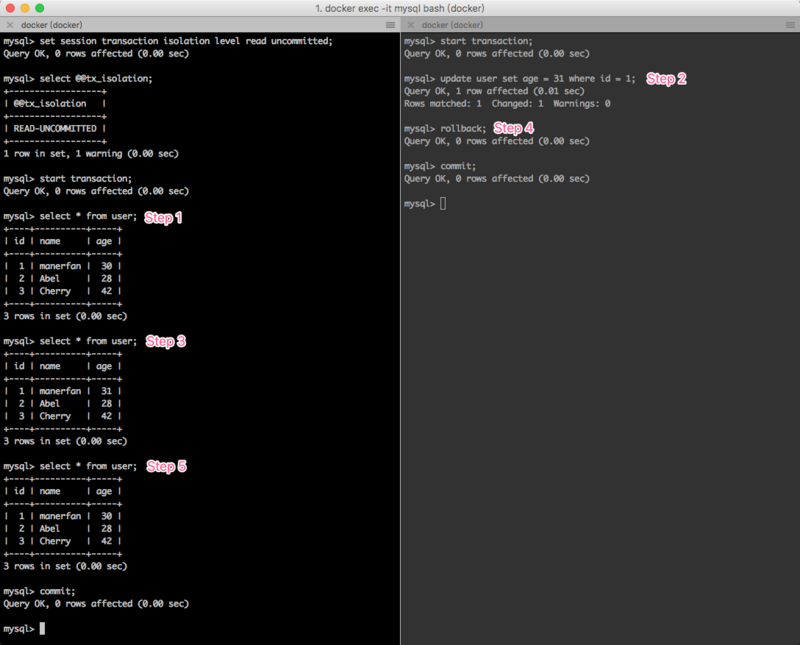
PRIMARY **KEY** (`id`)

) **ENGINE**=**InnoDB** **DEFAULT** **CHARSET**=utf8;

插入测试数据

**INSERT** **INTO** **user** (**name**, age) **VALUES** ('manerfan', 30), ('Abel', 28), ('Cherry', 42);

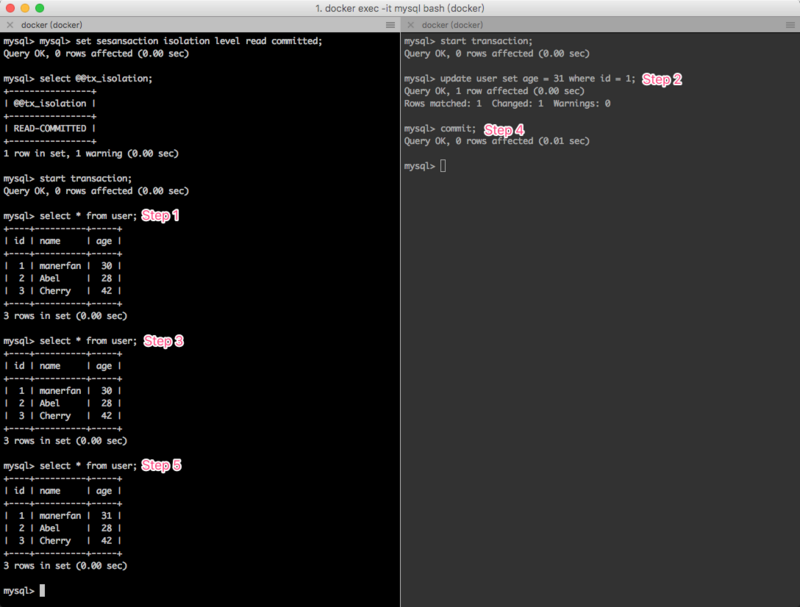
#### Read Uncommitted



Step 1: 设置A的隔离级别为Read Uncommitted，开启事务并读取数据  
Step 2: B开启事务，修改数据，但**不提交**  
Step 3: A读取数据，发现**数据已变**  
Step 4: B回滚，但**不提交**  
Step 5: A读取数据，发现**数据恢复**

A事务中可以读取到B事务未修改的数据，发生**赃读**

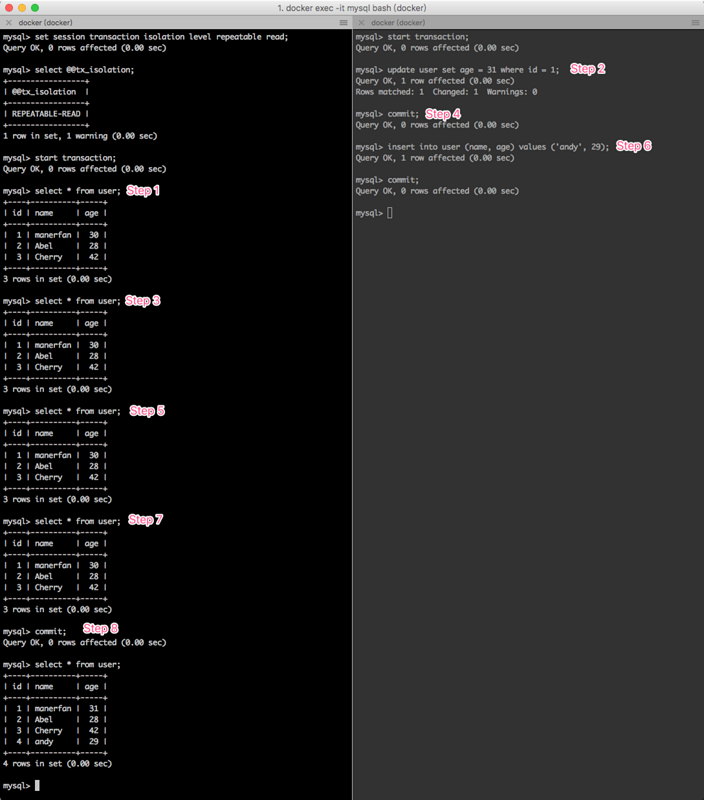
#### Read Committed



Step 1: 设置A的隔离级别为Read Committed，开启事务并读取数据  
Step 2: B开启事务，修改数据，但**不提交**  
Step 3: A读取数据，发现**数据未变**  
Step 4: B**提交事务**  
Step 5: A读取数据，发现**数据改变**

已提交读隔离级别解决了脏读的问题，但是出现了不可重复读的问题，即事务A在两次查询的数据不一致，因为在两次查询之间事务B更新了一条数据。

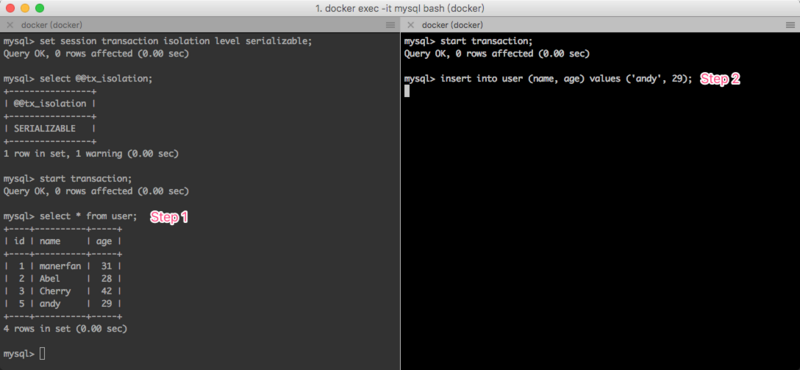
#### Repeatable Read



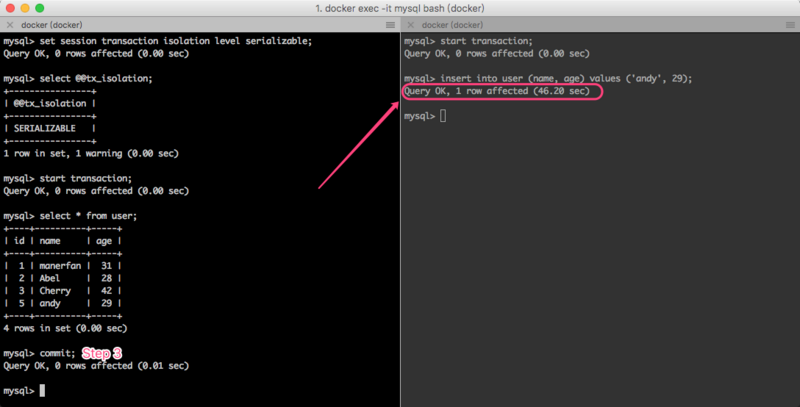
Step 1: 设置A的隔离级别为Repeatable Read，开启事务并读取数据  
Step 2: B开启事务，修改数据，但**不提交**  
Step 3: A读取数据，发现**数据未变**  
Step 4: B**提交事务**  
Step 5: A读取数据，发现**数据依然未变**，解决了不可重复读  
Step 6: B**插入新数据**，并**提交**  
Step 7: A读取数据，发现**数据还是未变**，出现**幻读**  
Step 8: A提交事务，再次读取数据，发现**数据改变**

Repeatable Read隔离级别只允许读取已提交记录，而且在一个事务两次读取一个记录期间，其他事务的更新不会影响该事务。但该事务不要求与其他事务可串行化，可能会发生幻读。

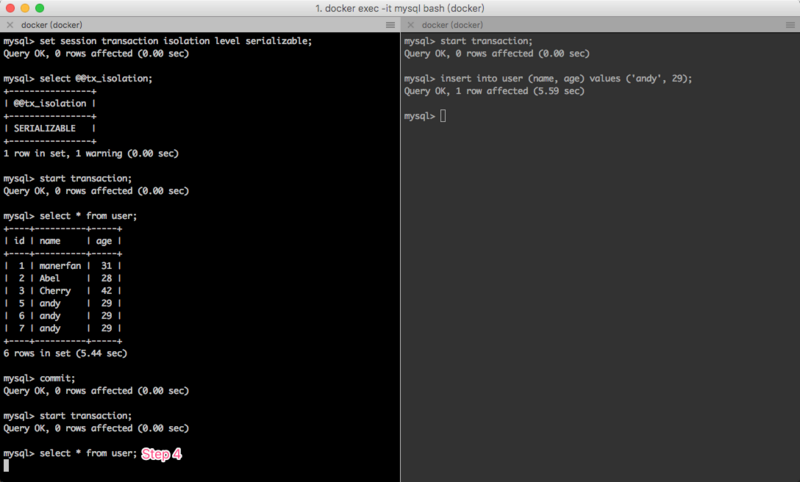
#### Serializable



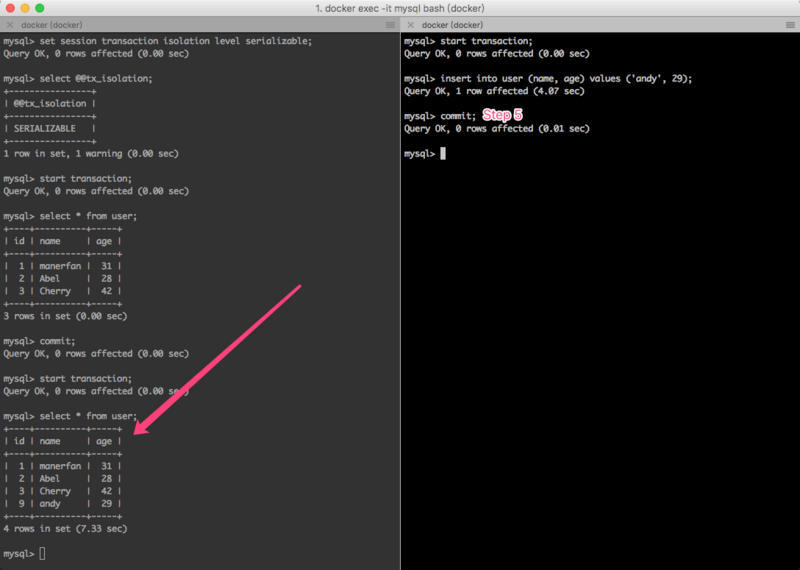
Step 1: 设置A的隔离级别为Serializable，开启事务并读取数据  
Step 2: B开启事务，修改数据，B**事务阻塞**，A的事务尚未提交，只能等待



Step 3: A事务提交，B事务插入成功，但**B事务不提交**



Step 4: A事务查询，发现A**事务阻塞**，B的事务尚未提交，只能等待



Step 5: B事务提交，A事务查询成功

Serializable隔离级别完全锁定字段，若一个事务来查询同一份数据就必须等待，直到前一个事务完成并解除锁定为止。

## 乐观锁与悲观锁

**乐观锁(Optimistic Lock)**，是指操作数据库时(更新操作)，总是认为这次的操作不会导致冲突，不到万不得已不去拿锁，在更新时采取判断是否冲突，适用于读操作远多于更新操作的情况。  
乐观锁并没有被数据库实现，需要自行实现，通常的实现方式为在表中增加版本version字段，更新时判断库中version与取出时的version值是否相等，若相等则执行更新并将version加1，若不相等则说明数据被其他线程(进程)修改，放弃修改。

**select** (age, **version**) **from** **user** **where** **id** = #{id};

# 其他操作

**update** **user** **set** age = 18, **version** = **version** + 1 **where** **id** = #{id} and version = #{version}

**悲观锁(Pessimistic Lock）**，是指操作数据库时(更新操作)，总是认为这次的操作会导致冲突，每次都要通过获取锁才能进行数据操作，因此要先确保获取锁成功再进行业务操作。  
悲观锁需要数据库自身提供支持，MySql提供了共享锁和排他锁来实现对数据行的锁定，两种锁的介绍如下介绍。

## MySql InnoDB引擎 锁

InnoDB实现了以下两种类型的行锁：

* 共享锁 (S): 允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁
* 排他锁 (X): 允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁



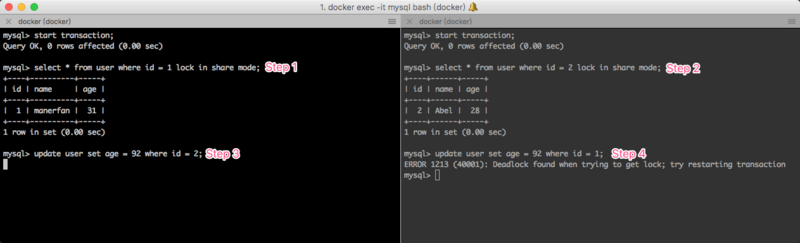
对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X）  
对于普通SELECT语句，InnoDB不会加任何锁

事务可以通过以下语句显式地给记录集加共享锁或排他锁：

* 共享锁: SELECT \* FROM table\_name WHERE ... LOCK IN SHARE MODE，等同读锁
* 排他锁: SELECT \* FROM table\_name WHERE ... FOR UPDATE，等同写锁

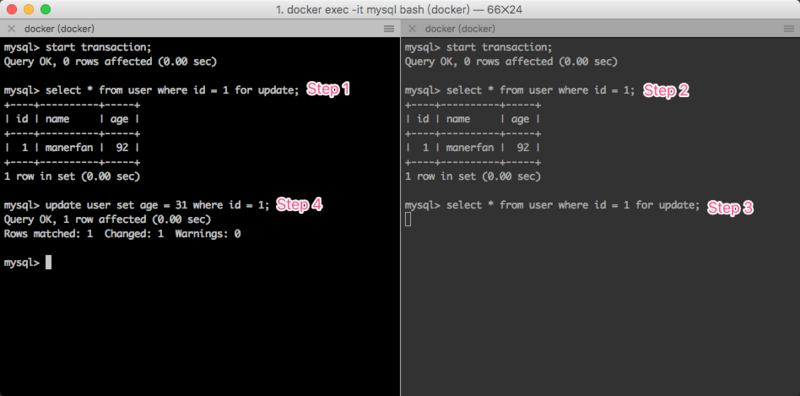
用SELECT ... IN SHARE MODE获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行UPDATE或者DELETE操作。但是如果当前事务也需要对该记录进行更新操作，则很有可能造成死锁，对于锁定行记录后需要进行更新操作的应用，应该使用SELECT... FOR UPDATE方式获得排他锁。

SELECT ... IN SHARE MODE模式

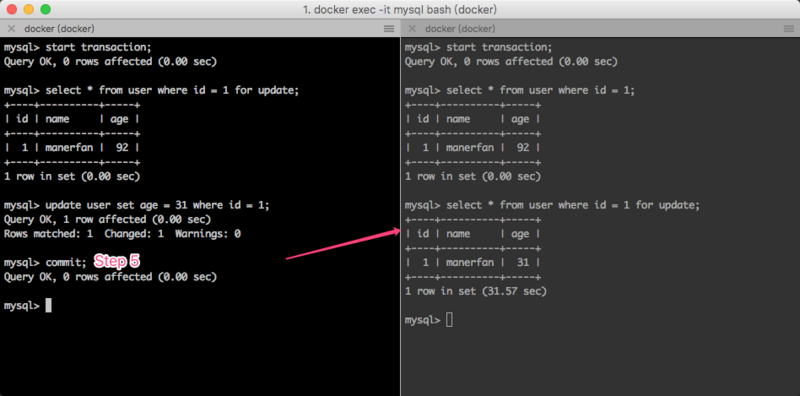


Step 1: A查询id为1的数据并加**共享锁**  
Step 2: B查询id为2的数据并加**共享锁**  
Step 3: A更新id为2的数据，由于共享锁与排他锁冲突而**阻塞**  
Step 4: B更新id为1的数据，由于A与B互相等待对方释放锁而抛出死锁异常

SELECT... FOR UPDATE模式

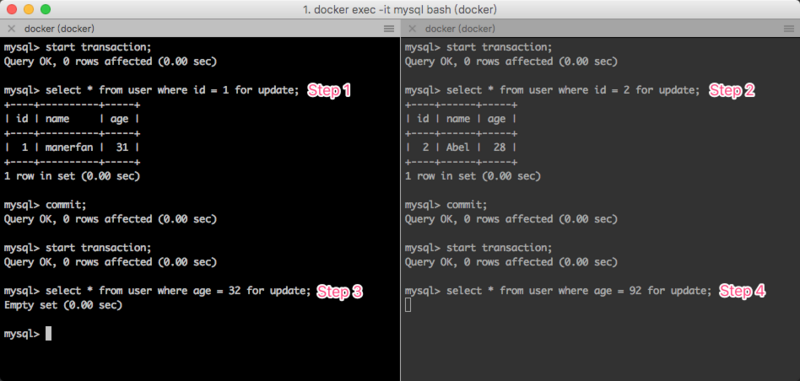


Step 1: A查询id为1的数据并加**排他锁**  
Step 2: B查询id为1的数据不加任何锁，成功  
Step 3: B查询id为1的数据并加**排他锁**，**阻塞**  
Step 4: A更新id为1的数据(数据库自动加排他锁)，成功



Step 5: A提交事务，B获取锁查询成功

InnoDB行锁是通过**给索引上的索引项加锁**来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！  
**在实际应用中，要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。**



Step 1: A查询id为1的数据并加**排他锁**  
Step 2: B查询id为2的数据并加**排他锁**，**成功**  
Step 3: A开启新的事物，查询age为32的数据并加**排他锁**  
Step 4: B开启新的事物，查询age为92的数据并加**排他锁**，**阻塞**，B与A查询的数据并不是同一行，但B阻塞，说明A的排他锁为表锁非行锁

## Spring Transaction的事务传播行为

Spring的 @Transactional 提供了设置事务隔离级别及事务传播行为的方式

**Isolation**中定义了DEFAULT及以上介绍的四中隔离级别，这里不再赘述

**package** org.springframework.transaction.annotation;

**public** **enum** **Isolation** {

DEFAULT, READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE;

}

**Propagation**中定义了其中传播行为

**package** org.springframework.transaction.annotation;

**public** **enum** **Propagation** {

REQUIRED, SUPPORTS, MANDATORY, REQUIRES\_NEW, NOT\_SUPPORTED, NEVER, NESTED;

}

* **PROPAGATION\_REQUIRED**：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务。
* **PROPAGATION\_SUPPORTS**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。
* **PROPAGATION\_MANDATORY**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。
* **PROPAGATION\_REQUIRES\_NEW**：创建新事务，无论当前存不存在事务，都创建新事务。
* **PROPAGATION\_NOT\_SUPPORTED**：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
* **PROPAGATION\_NEVER**：以非事务方式执行，如果当前存在事务，则抛出异常。
* **PROPAGATION\_NESTED**：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。