首先，我们知道事务具有ACID四个特性。也即：原子性，一致性，隔离性，持久性

持久性

1. 定义

持久性是指事务一旦提交，它对数据库的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

2. 实现原理：redo log

redo log和undo log都属于InnoDB的事务日志。下面先聊一下redo log存在的背景。

InnoDB作为MySQL的存储引擎，数据是存放在磁盘中的，但如果每次读写数据都需要磁盘IO，效率会很低。为此，InnoDB提供了缓存(Buffer Pool)，Buffer Pool中包含了磁盘中部分数据页的映射，作为访问数据库的缓冲：当从数据库读取数据时，会首先从Buffer Pool中读取，如果Buffer Pool中没有，则从磁盘读取后放入Buffer Pool；当向数据库写入数据时，会首先写入Buffer Pool，Buffer Pool中修改的数据会定期刷新到磁盘中（这一过程称为刷脏）。

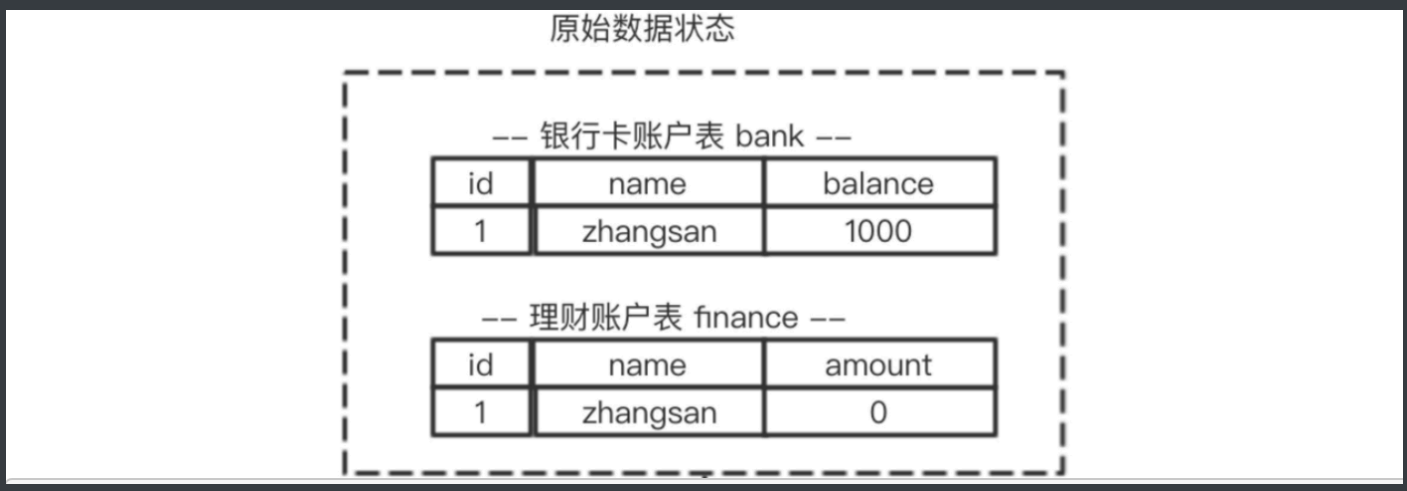
Buffer Pool的使用大大提高了读写数据的效率，但是也带了新的问题：如果MySQL宕机，而此时Buffer Pool中修改的数据还没有刷新到磁盘，就会导致数据的丢失，事务的持久性无法保证。

于是，redo log被引入来解决这个问题：当数据修改时，除了修改Buffer Pool中的数据，还会在redo log记录这次操作；当事务提交时，会调用fsync接口对redo log进行刷盘。如果MySQL宕机，重启时可以读取redo log中的数据，对数据库进行恢复。redo log采用的是WAL（Write-ahead logging，预写式日志），所有修改先写入日志，再更新到Buffer Pool，保证了数据不会因MySQL宕机而丢失，从而满足了持久性要求。

既然redo log也需要在事务提交时将日志写入磁盘，为什么它比直接将Buffer Pool中修改的数据写入磁盘(即刷脏)要快呢？主要有以下两方面的原因：

1、刷脏是随机IO，因为每次修改的数据位置随机，但写redo log是追加操作，属于顺序IO。

2、刷脏是以数据页（Page）为单位的，MySQL默认页大小是16KB，一个Page上一个小修改都要整页写入；而redo log中只包含真正需要写入的部分，无效IO大大减少。



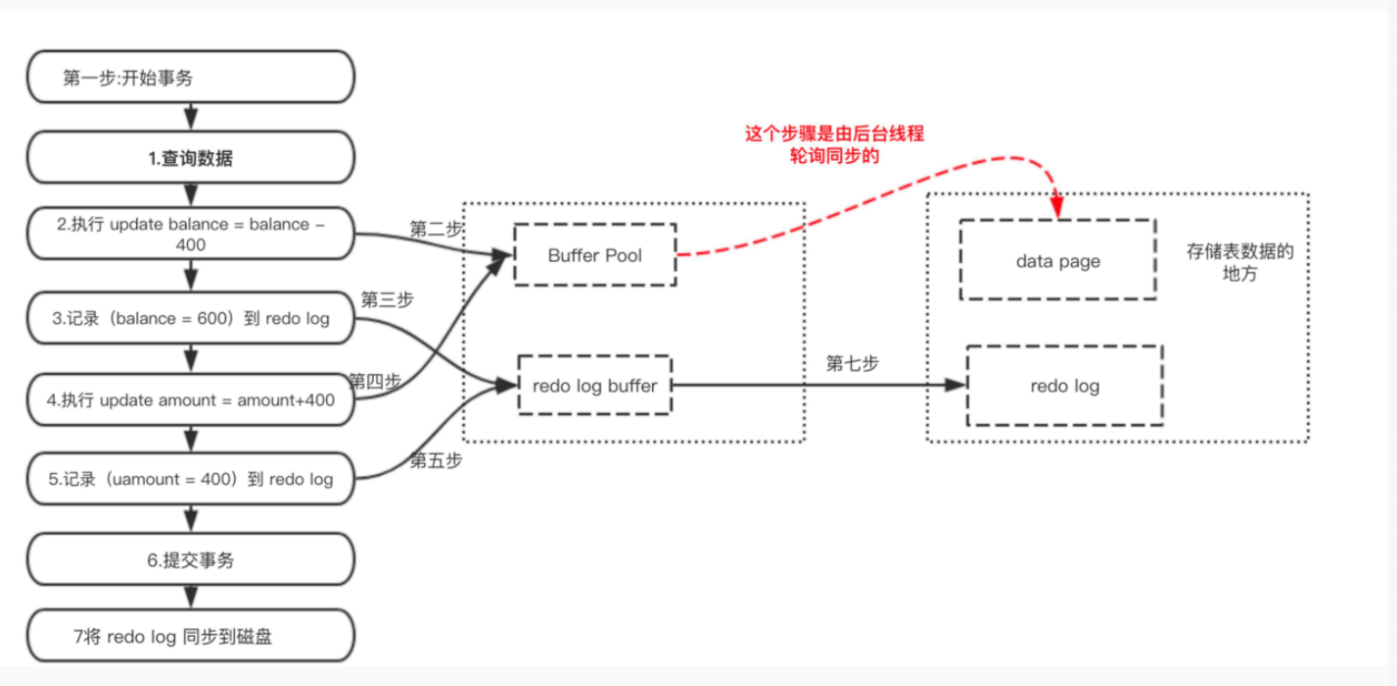
start transaction;

select balance from bank where name="zhangsan";

// 生成 重做日志 balance=600 update bank set balance = balance - 400;

// 生成 重做日志 amount=400 update finance set amount = amount + 400;

commit;



总结：redo log是用来恢复数据的，用于保障已提交事务的持久化特性

原子性

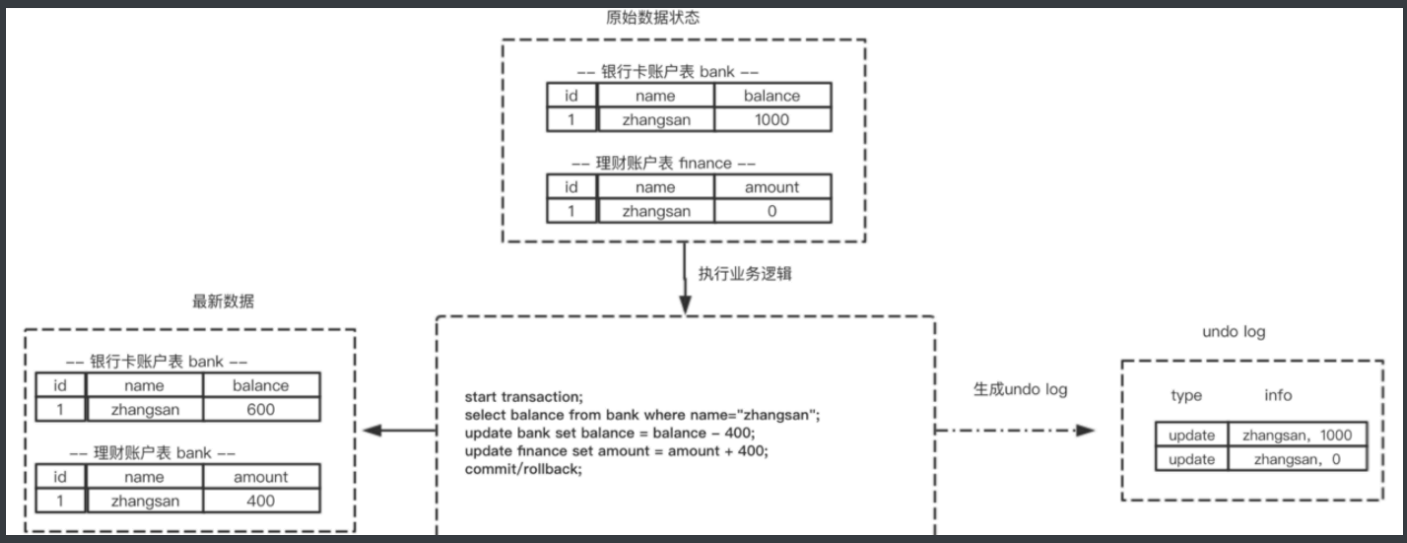
1. 定义

原子性是指一个事务是一个不可分割的工作单位，其中的操作要么都做，要么都不做；如果事务中一个sql语句执行失败，则已执行的语句也必须回滚，数据库退回到事务前的状态。

2. 实现原理：undo log

实现原子性的关键，是当事务回滚时能够撤销所有已经成功执行的sql语句。InnoDB实现回滚依据的是undo log：当事务对数据库进行修改时，InnoDB会生成对应的undo log；如果事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。

undo log属于逻辑日志，它记录的是sql执行相关的信息。当发生回滚时，InnoDB会根据undo log的内容做与之前相反的工作：对于每个insert，回滚时会执行delete；对于每个delete，回滚时会执行insert；对于每个update，回滚时会执行一个相反的update，把数据改回去。



总结：undo log是用来回滚数据的，用于保障未提交事务的原子性。

隔离性

SQL 规范定义了以上四种隔离级别，但是并没有给出如何实现四种隔离级别，不同数据库的实现方式和使用方式也并不相同。而 SQL 隔离级别的标准是依据基于锁的实现方式来制定的，因此有必要先了解一下传统的基于锁的隔离级别是如何实现的。

1、传统隔离级别的实现

传统的隔离级别是基于锁实现的，我们先来了解一下锁。

锁

传统的锁模式有两种：

- 共享锁（ Shared Locks ）：简称 S锁 ，事务对一条记录进行读操作时，需要先获取该记录的共享锁。
- 排他锁（ Exclusive Locks ）：简称 X锁 ，事务对一条记录进行写操作时，需要先获取该记录的排他锁。

需要注意的是，加了共享锁的记录，其他事务也可以获得该记录的共享锁，但是无法获取该记录的排他锁，即 S锁 和 S锁 是兼容的， S锁 和 X锁 是不兼容的；而加了排他锁的记录，其他事务既无法获取该记录的共享锁也无法获取排他锁，即 X锁 和 X锁 也是不兼容的。

基于锁实现隔离级别

在基于锁的实现方式下，四种隔离级别的区别就在于加锁方式的区别：

- **读未提交**：事务对当前被**读取的数据（读取的瞬间，或者是读取前一瞬间）**加共享锁，一旦读完该行，立即释放该行的共享锁,由此导致不可重复读。事务对需要修改的数据（修改的瞬间，或者是说修改前一瞬间） 对其加共享锁（加锁后其他事务不能更改，但是可以读取，由此导致“脏读”），直到事务结束才释放。事务结束包括正常结束（COMMIT）和非正常结束（ROLLBACK）。

- **读已提交**：事务对当前被**读取的数据（读取的瞬间，或者是读取前一瞬间）**加共享锁，一旦读完该行，立即释放该行的共享锁。

  事务对需要修改的数据（修改的瞬间，或者是说修改前一瞬间） 对其加排他锁,直到事务结束才释放,防止其他事务读取未提交的数据，这样，也就避免了 “脏读” 的情况。事务结束包括正常结束（COMMIT）和非正常结束（ROLLBACK）。
- **可重复读**：事务对当前被**读取的数据（读取的瞬间，或者是读取前一瞬间）**加共享锁\，直到**事务结束才释放**，这样保证了可重复读（既其他的事务能读取该数据，但是不能修改该数据）。事务对需要修改的数据（修改的瞬间，或者是说修改前一瞬间） 对其加排他锁,直到事务结束才释放,防止其他事务读取未提交的数据，这样，也就避免了 “脏读” 的情况。事务结束包括正常结束（COMMIT）和非正常结束（ROLLBACK）。
- **串行化**：读操作和写操作都加 `X锁` 且直到事务提交后才释放，粒度为表锁，也就是严格串行。

不同数据库对于 SQL 标准中规定的隔离级别支持是不一样的，数据库引擎实现隔离级别的方式虽然都在尽可能地贴近标准的隔离级别规范，但和标准的预期还是有些不一样的地方。

`MySQL （ InnoDB ）` 支持的4种隔离级别，与标准的各级隔离级别允许出现的问题有些出入，比如 `MySQL` 在可重复读隔离级别下可以防止幻读的问题出现。

MVCC的实现原理

为了方便描述，首先我们创建一个表 `book` ，就三个字段，分别是主键 `book_id` ，名称 `book_name` ，库存 `stock` 。然后向表中插入一些数据：

```
INSERT INTO book VALUES(1, '数据结构', 100);
INSERT INTO book VALUES(2, 'C++指南', 100);
INSERT INTO book VALUES(3, '精通Java', 100);
```

版本链

对于使用 InnoDB 存储引擎的表，其聚簇索引记录中包含了两个重要的隐藏列：

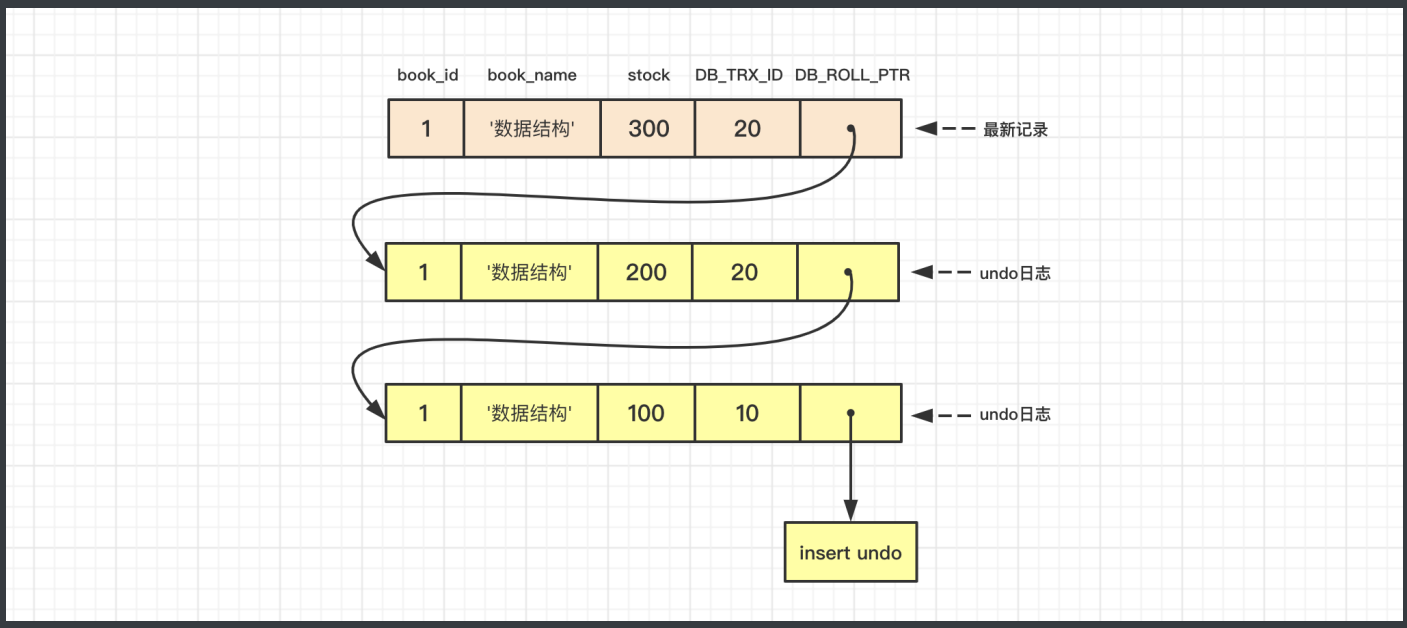
- 事务ID（ `DB_TRX_ID` ）：每当事务对聚簇索引中的记录进行修改时，都会把当前事务的事务id记录到 `DB_TRX_ID` 中。
- 回滚指针（ `DB_ROLL_PTR` ）：每当事务对聚簇索引中的记录进行修改时，都会把该记录的旧版本记录到 undo 日志中，通过 `DB_ROLL_PTR` 这个指针可以用来获取该记录旧版本的信息。

如果在一个事务中多次对记录进行修改，则每次修改都会生成 undo 日志，并且这些 undo 日志通过 `DB_ROLL_PTR` 指针串联成一个版本链，版本链的头结点是该记录最新的值，尾结点是事务开始时的初始值。

例如，我们在表 `book` 中做以下修改：

```
BEGIN;

UPDATE book SET stock = 200 WHERE id = 1;

UPDATE book SET stock = 300 WHERE id = 1;
```

那么 id=1 的记录此时的版本链就如下图所示：



ReadView

对于使用 Read Uncommitted 隔离级别的事务来说，只需要读取版本链上最新版本的记录即可；对于使用 Serializable 隔离级别的事务来说， InnoDB 使用加锁的方式来访问记录。而 Read Committed 和 Repeatable Read 隔离级别来说，都需要读取已经提交的事务所修改的记录，也就是说如果版本链中某个版本的修改没有提交，那么该版本的记录时不能被读取的。所以需要确定在 Read Committed 和 Repeatable Read 隔离级别下，版本链中哪个版本是能被当前事务读取的。于是 ReadView 的概念被提出以解决这个问题。

ReadView 相当于某个时刻表记录的一个快照，在这个快照中我们能获取到与当前记录相关的事务中，哪些事务是已提交的**稳定事务**，哪些是**正在活跃的事务**，哪些是**生成快照之后才开启的事务**。由此我们就能根据**可见性比较算法**判断出版本链中能被读取的最新版本记录。

可见性比较算法是基于事务ID的比较算法。首先我们需要知道的一个事实是：事务 id 是递增分配的。从 ReadView 中我们能获取到生成快照时刻系统中活跃的事务中最小和最大的事务 id ，这样我们就得到了一个活跃事务 id 的范围，我们可称之为 ACTIVE_TRX_ID_RANGE 。那么小于这个范围的事务id对应的事务都是已提交的稳定事务，大于这个范围的事务都是在快照生成之后才开启的事务，而在 ACTIVE_TRX_ID_RANGE 范围内的事务中除了正在活跃的事务，也都是已提交的稳定事务。

有了以上信息之后，我们顺着版本链从头结点开始查找最新的可被读取的版本记录：

1、首先判断版本记录的 DB_TRX_ID 字段与生成 ReadView 的事务对应的事务ID是否相等。如果相等，那就说明该版本的记录是在当前事务中生成的，自然也就能够被当前事务读取；否则进行第2步。

2、如果版本记录的 DB_TRX_ID 字段小于范围 ACTIVE_TRX_ID_RANGE ，表明该版本记录是已提交事务修改的记录，即对当前事务可见；否则进行下一步。

3、如果版本记录的 DB_TRX_ID 字段位于范围 ACTIVE_TRX_ID_RANGE 内，如果该事务ID对应的不是活跃事务，表明该版本记录是已提交事务修改的记录，即对当前事务可见；如果该事务ID对应的是活跃事务，那么对当前事务不可见，则读取版本链中下一个版本记录，重复以上步骤，直到找到对当前事务可见的版本。

如果某个版本记录经过以上步骤判断确定其对当前事务可见，则查询结果返回此版本记录；否则读取下一个版本记录继续按照上述步骤进行判断，直到版本链的尾结点。如果遍历完版本链没有找到对当前事务可见的版本，则查询结果为空。

在 MySQL 中， Read Committed 和 Repeatable Read 隔离级别下的区别就是它们生成 ReadView 的时机不同。

MVCC实现不同隔离级别

之前说到 ReadView 的机制只在 Read Committed 和 Repeatable Read 隔离级别下生效，所以只有这两种隔离级别才有 MVCC 。
在 Read Committed 隔离级别下，每次读取数据时都会生成 ReadView ；而在 Repeatable Read 隔离级别下只会在事务首次读取数据时生成 ReadView ，之后的读操作都会沿用此 ReadView 。

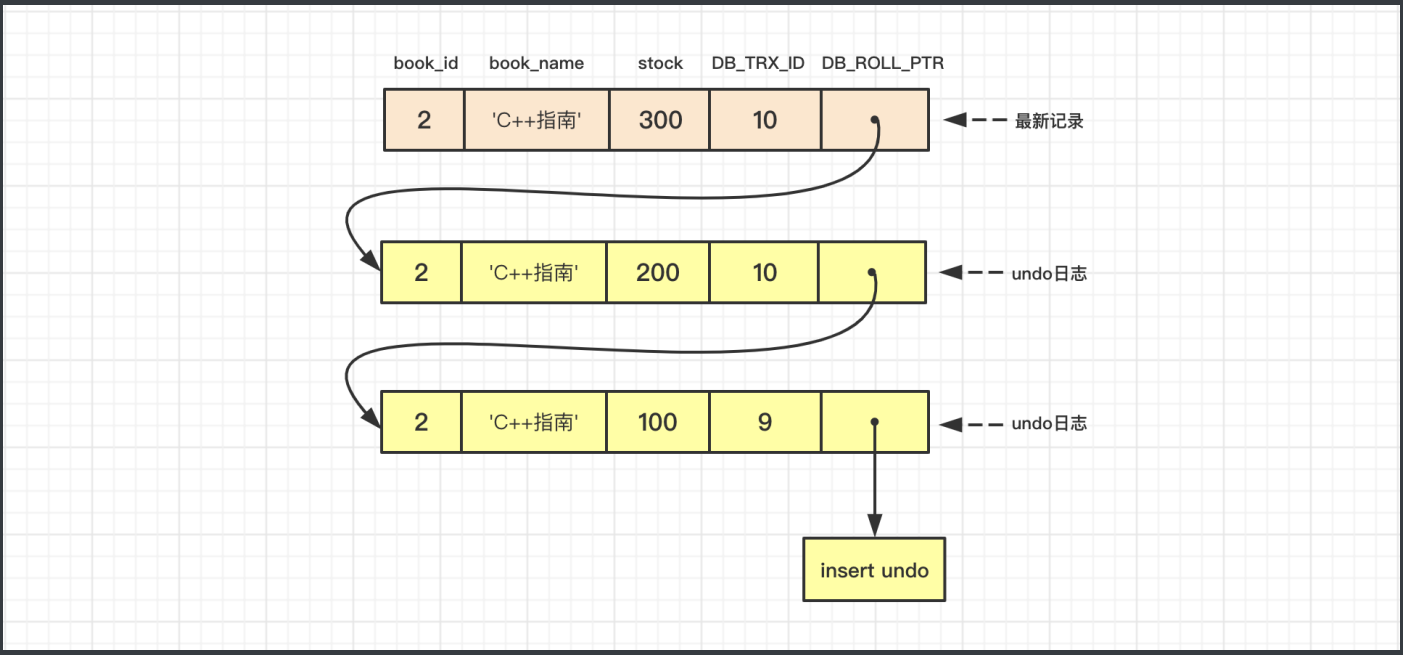
下面我们通过例子来看看 Read Committed 和 Repeatable Read 隔离级别下 MVCC 的不同表现。我们继续以表 book 为例进行演示。

Read Committed隔离级别分析

假设在 Read Committed 隔离级别下，有如下事务在执行，事务 id 为10：

```
BEGIN; // 开启Transaction 10

UPDATE book SET stock = 200 WHERE id = 2;

UPDATE book SET stock = 300 WHERE id = 2;
```

此时该事务尚未提交， id 为2的记录版本链如下图所示：



然后我们开启一个事务对 id 为2的记录进行查询：

```
BEGIN;

SELECT * FROM book WHERE id = 2;
```

当执行 SELECT 语句时会生成一个 ReadView，该 ReadView 中的 ACTIVE_TRX_ID_RANGE 为 [10, 11)，当前事务 ID creator_trx_id 为 0 （因为事务中当执行写操作时才会分配一个单独的事务 id，否则事务 id 为 0）。按照我们之前所述 ReadView 的工作原理，我们查询到的版本记录为

```
+----------+-----------+-------+
| book_id  | book_name | stock |
+----------+-----------+-------+
| 2        | C++指南     |  100  |
+----------+-----------+-------+
```

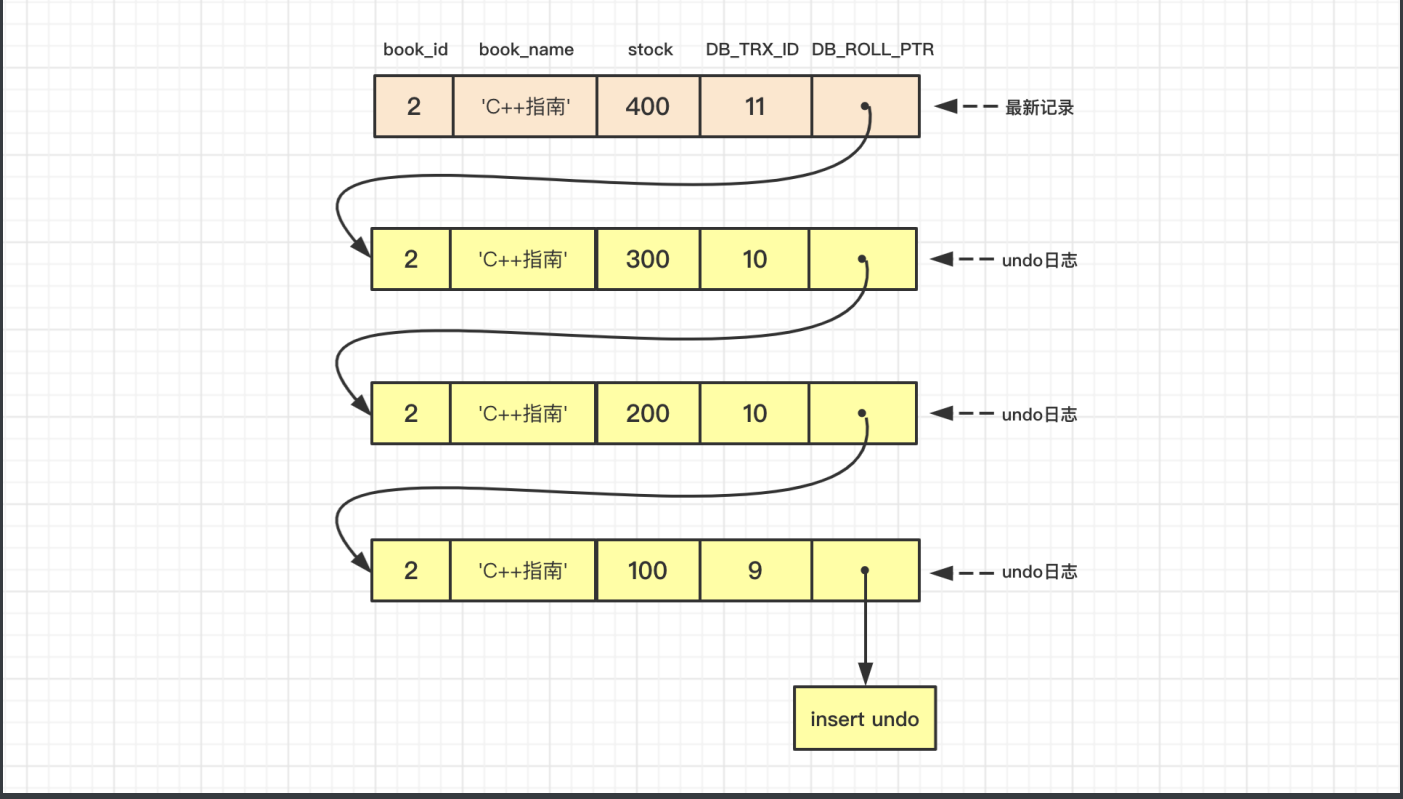
然后我们将事务 id 为10的事务提交：

```
BEGIN; // 开启Transaction 10

UPDATE book SET stock = 200 WHERE id = 2;

UPDATE book SET stock = 300 WHERE id = 2;

COMMIT;
```

同时开启执行另一事务 `id` 为 `11` 的事务，但不提交：

```
BEGIN; // 开启Transaction 11

UPDATE book SET stock = 400 WHERE id = 2;
```

此时 `id` 为2的记录版本链如下图所示：



然后我们回到刚才的查询事务中再次查询 `id` 为2的记录：

```
BEGIN;

SELECT * FROM book WHERE id = 2; // 此时Transaction 10 未提交

SELECT * FROM book WHERE id = 2; // 此时Transaction 10 已提交
```

当第二次执行 SELECT 语句时会再次生成一个 ReadView ，该 ReadView 中的 ACTIVE_TRX_ID_RANGE 为 [11, 12) ，当前事务ID creator_trx_id 依然为 0 。按照 ReadView 的工作原理进行分析，我们查询到的版本记录为

```
+----------+-----------+-------+
| book_id  | book_name | stock |
+----------+-----------+-------+
| 2        | C++指南    | 300   |
+----------+-----------+-------+
```

从上述分析可以发现，因为每次执行查询语句都会生成新的 ReadView ，所以在 Read Committed 隔离级别下的事务读取到的是查询时刻表中已提交事务修改之后的数据。

Repeatable Read隔离级别分析

我们在 Repeatable Read 隔离级别下重复上面的事务操作：

```
BEGIN; // 开启Transaction 20

UPDATE book SET stock = 200 WHERE id = 2;

UPDATE book SET stock = 300 WHERE id = 2;
```

此时该事务尚未提交，然后我们开启一个事务对 id 为2的记录进行查询：

```
BEGIN;

SELECT * FROM book WHERE id = 2;
```

当事务第一次执行 SELECT 语句时会生成一个 ReadView ，该 ReadView 中的 ACTIVE_TRX_ID_RANGE 为 [10, 11) ，当前事务ID creator_trx_id 为 0 。根据 ReadView 的工作原理，我们查询到的版本记录为

```
+----------+-----------+-------+
| book_id  | book_name | stock |
+----------+-----------+-------+
| 2        | C++指南    |  100  |
+----------+-----------+-------+
```

然后我们将事务 id 为20的事务提交：

```
BEGIN; // 开启Transaction 20

UPDATE book SET stock = 200 WHERE id = 2;

UPDATE book SET stock = 300 WHERE id = 2;

COMMIT;
```

同时开启执行另一事务 `id` 为21的事务，但不提交：

```
BEGIN; // 开启Transaction 21

UPDATE book SET stock = 400 WHERE id = 2;
```

然后我们回到刚才的查询事务中再次查询 `id` 为2的记录：

```
BEGIN;

SELECT * FROM book WHERE id = 2; // 此时Transaction 10 未提交

SELECT * FROM book WHERE id = 2; // 此时Transaction 10 已提交
```

当第二次执行 `SELECT` 语句时不会生成新的 `ReadView` ，依然会使用第一次查询时生成 `ReadView` 。因此我们查询到的版本记录跟第一次查询到的结果是一样的：

```
+----------+-----------+-------+
| book_id  | book_name | stock |
+----------+-----------+-------+
| 2        | C++指南    |  100  |
+----------+-----------+-------+
```

从上述分析可以发现，因为在 `Repeatable Read` 隔离级别下的事务只会在第一次执行查询时生成 `ReadView` ，该事务中后续的查询操作都会沿用这个 `ReadView` ，因此此隔离级别下一个事务中多次执行同样的查询，其结果都是一样的，这样就实现了可重复读。

一致性

MySQL事务一致性通过原子性、持久性、隔离性来实现。