# MySQL——事务隔离级别和MVCC





# 一、事务隔离级别

- *什么是事务的隔离性?*

> 如果**多个事务**访问了**同样的一条数据**，那么会造成数据的一致性问题。这就要求我们使用某种手段来强制让这些事务按照顺序一个一个单独地执行，或者最终执行的效果和单独执行一样。也就是

> 说我们希望让这些事务“隔离”地执行，互不干涉。这也就是事务的隔离性。

- 当多个事务对同一条数据进行写操作的时候，就会涉及到一致性的问题。我们可以通过串行化来解决。但是却会造成一定的性能损失。我们思考，*是否可以牺牲一部分隔离性来换取性能上的提升呢?* 是的，当然可以。不过我们首先需要搞明白，*多个事务在不进行串行化执行的情况下，到底会出现哪些一致性问题?*

# 1.1> 事务并发执行时遇到的问题

## 1.1.1> 脏写

- *什么是脏写?*

> 如果一个事务T2**修改了**另一个**未提交事务**T1**修改过**的数据，这就意味着发生了脏写现象。

- 脏写引发的**一致性**问题:

> 我们希望x值和y值始终相同。下面是事务T1和事务T2对x值和y值的操作:
> w1[x=1] w2[x=2] w2[y=2] c2 w1[y=1] c1
> 最终导致了x=2 y=1，x值和y值不相同，破坏了一致性需求。

- 脏写引发的**原子性和持久性**问题:

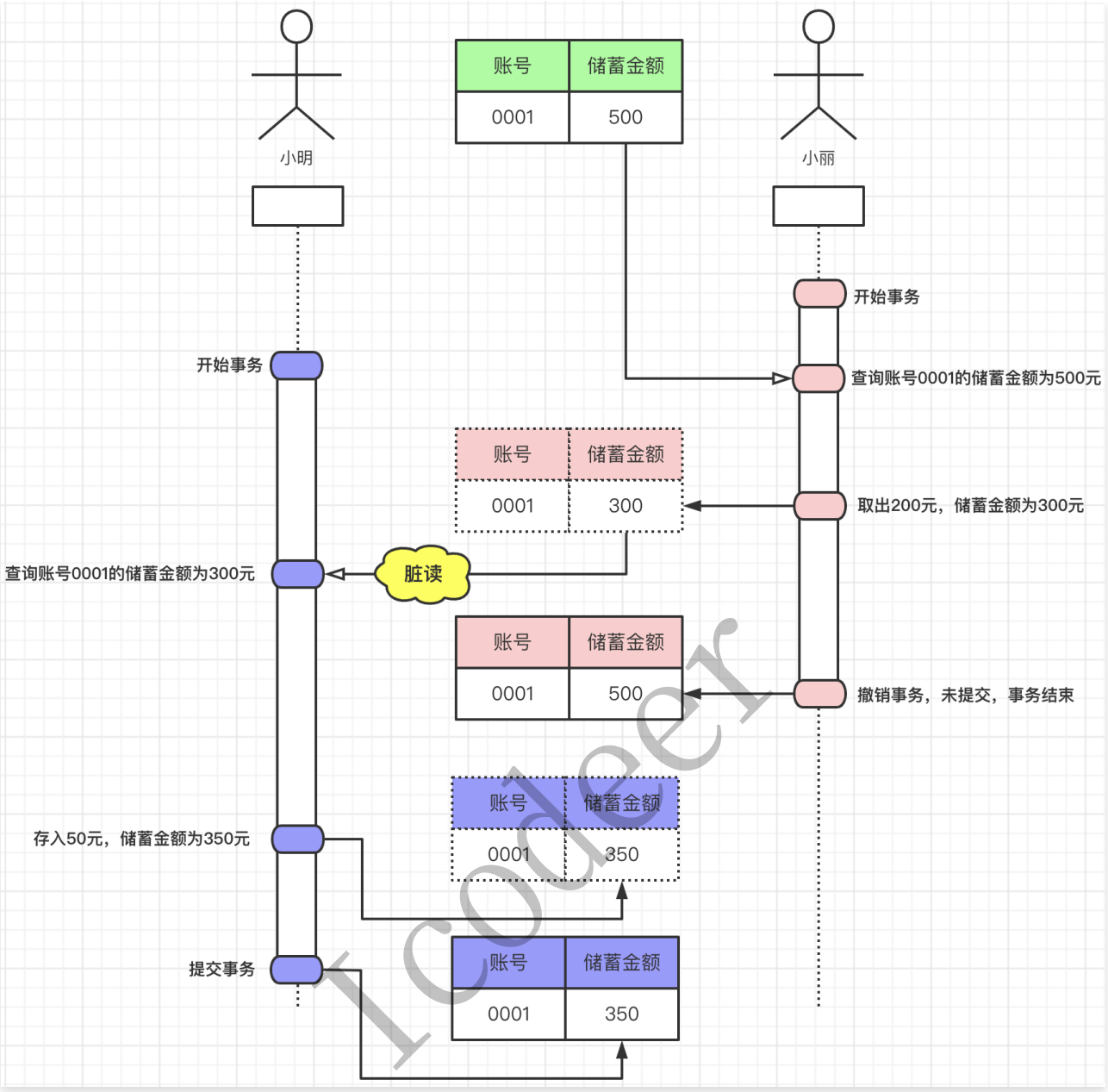
> 比方说有x=0和y=0这两个数据项，下面是事务T1和事务T2对x值和y值的操作:
> w1[x=2] w2[x=3] w2[y=3] c2 a1
> 由于T1执行了回滚操作，即x=0（将x置为最初状态），那么相当于对T2对数据库所做的修改进行了**部分回滚**（即：T2只回滚了对x做的修改，而不回滚对y做的修改），那么这就影响到了事务的**原子性**。而T2已经提交了，但是却被T1的回滚造成了自己修改的数据也被回滚了，破坏了T2事务的**持久性**。

## 1.1.2> 脏读

- *什么是脏读?*

> 脏读就是指当一个事务T1正在访问数据，并且对数据进行了修改，而这种**修改还没有提交**到数据库中，这时，另外一个事务T2也访问这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是脏数据(`Dirty Data`)，依据脏数据所做的操作可能是不正确的。

- 操作流程如下图所示:




- 脏读引发的**一致性**问题：

> 事务T1和事务T2访问 `x` 和 `y` 这两个值，我们一致性需求就是让x值和y值始终相同，x和y的初始值都是0。现在并发执行事务T1和T2如下所示：
> w1[x=1] r2[x=1] r2[y=0] c2 w1[y=1] c1
> 很显然T2是个只读事务，读取到了事务T1未提交事务的值，所以T2读到的x=1，y=0，不符合x=y的一致性。数据库的不一致状态是不应该暴露给用户的。

- 脏读的严格解释

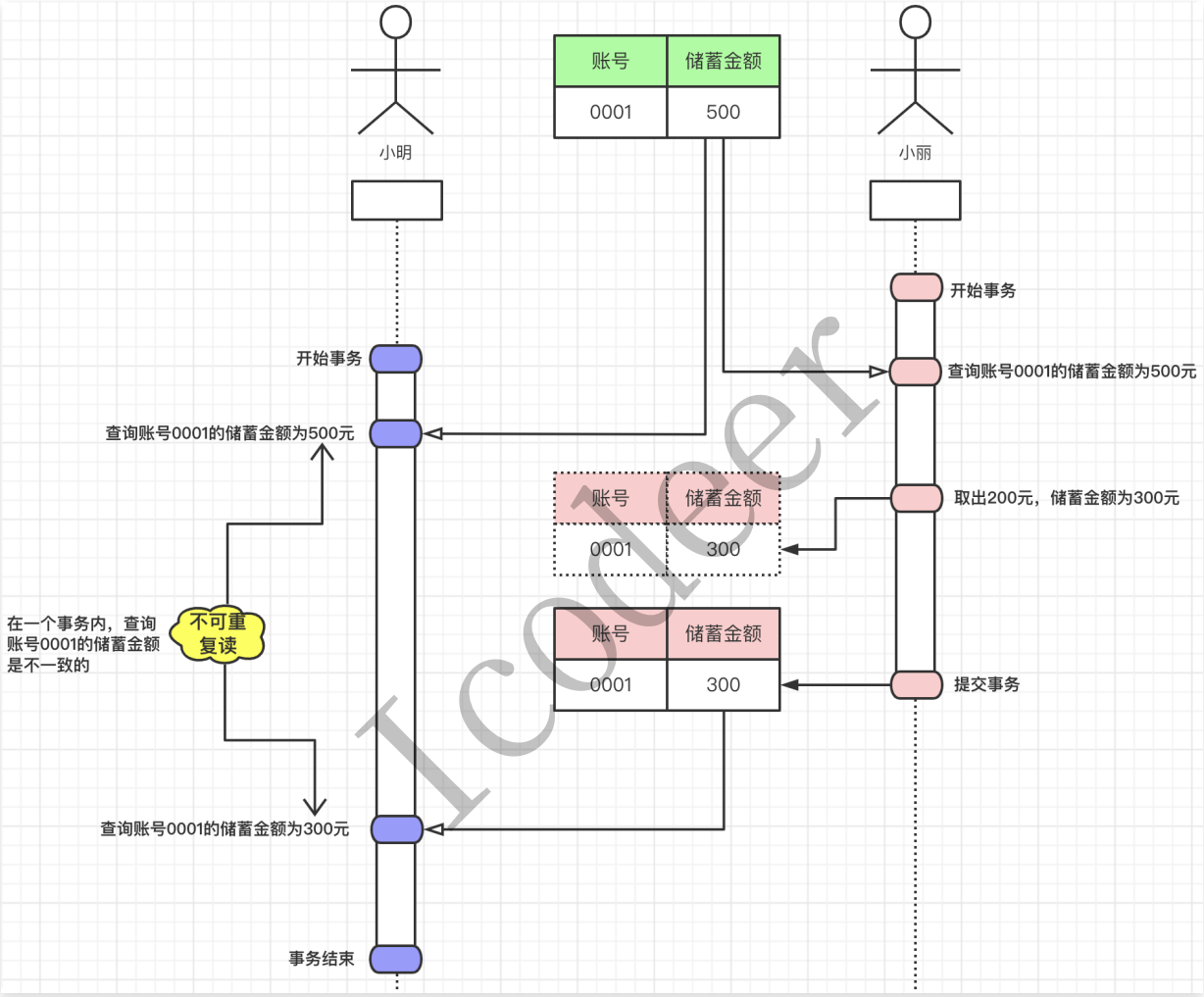
> w1[x] ... r2[x] ... (a1 and c2 in any order)
> 也就是T1先修改了数据项x的值，然后T2又读取到了**未提交事务**T1针对数据项x修改后的值，之后T1回滚而T2提交。这就意味着T2**读到了一个根本不存在的值**。

### 1.1.3> 不可重复读

- *什么是不可重复读?*

> 指在一个事务T1内，多次读同一数据。在这个事务T1还没有结束时，另外一个事务T2修改并提交了该同一数据。那么，事务T1两次读到的数据可能是不一样的。这样就发生了**在一个事务内两次读到的数据是不一样的**，因此称为是不可重复读。

- 操作流程如下图所示：




- 不可重复读的**一致性**问题：

> 事务T1和事务T2访问x和y这两个值，我们一致性需求就是让x值和y值始终相同，x和y的初始值都是0。现在并发执行事务T1和T2如下所示：
> r1[x=0] w2[x=1] w2[y=1] c2 r1[y=1] c1
> 很显然T1是个只读事务，最终读取到的是x=0，y=1，很显然这是一个不一致的状态，这种不一致的状态是不应该暴露给用户的。

- 不可重复读的严格解释

> r1[x] ... w2[x] ... c2 ... r1[x] ... c1

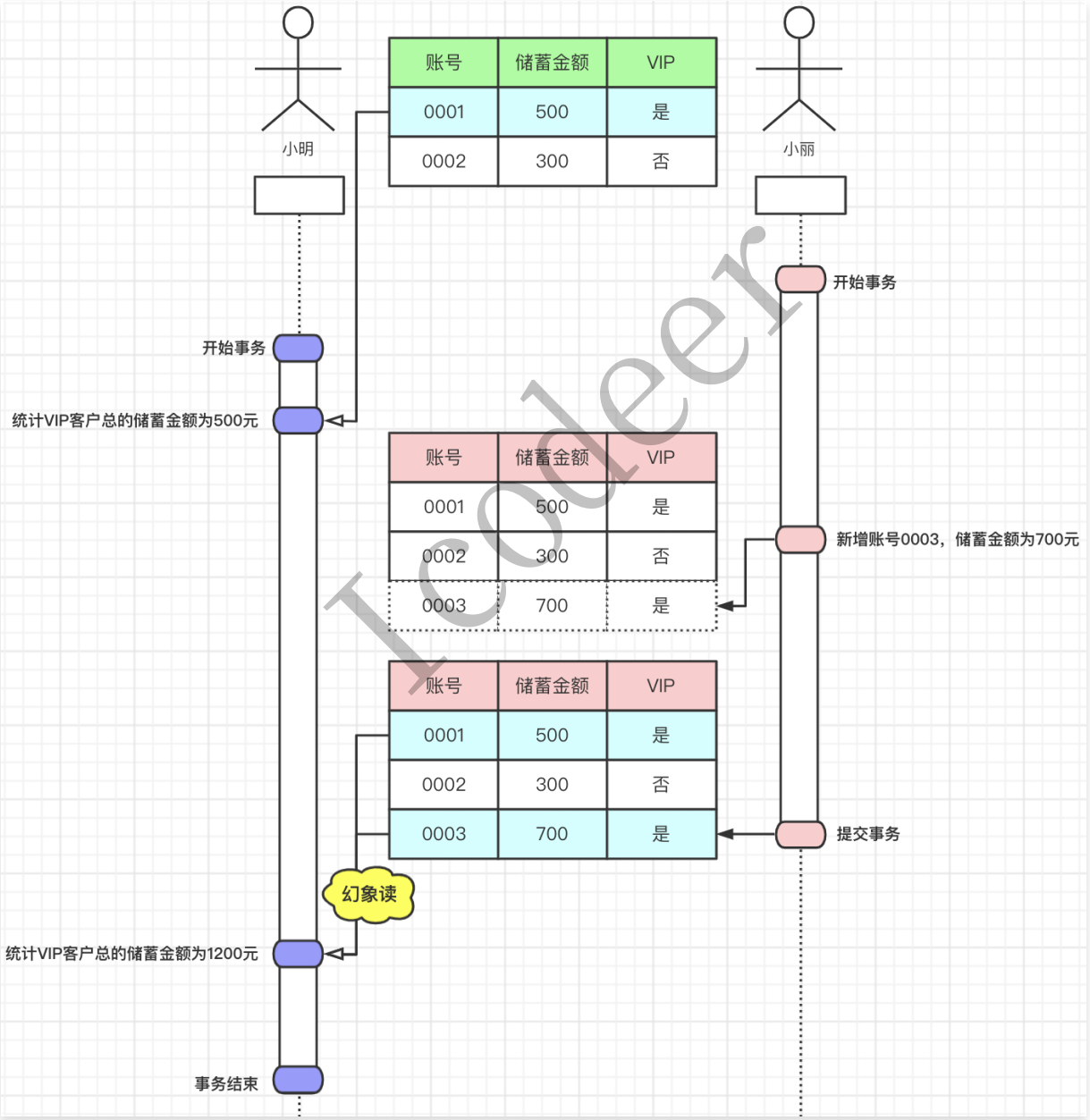
> 也就是T1先读取了数据项x的值，然后T2又修改了x的值，之后T2提交事务，然后T1再次读取数据项x的值时会得到与第一次读取时不同的值。

### 1.1.4> 幻读

- 什么是幻读

> 如果一个事务T1先根据**某些搜索条件**查询出一些记录，在该事务未提交时，另一个事务T2操作了一些符合那些搜索条件的记录（insert、delete、update），就意味着发生了幻读现象。

- 操作流程如下图所示：




- 幻读的**一致性**问题：

> r1[P] ... w2[y in P] ... c2 ... r1[P] ... c1

T1先读取符合搜索条件P的记录，然后T2写入了符合搜索条件P的记录。之后T1再读取符合搜索条件P的记录时，会发现两次读取的记录是不一样的。

## 1.2> SQL标准中的4种隔离级别

- 四种隔离级别如下表所示：

| 隔离级别 | 脏读 | 不可重复读 | 幻读 |
| --- | --- | --- | --- |
| READ UNCOMMITTED | 可能 | 可能 | 可能 |
| READ COMMITTED | 不可能 | 可能 | 可能 |
| REPEATABLE READ | 不可能 | 不可能 | 可能 |
| SERIALIZABLE | 不可能 | 不可能 | 不可能 |

- 由于无论哪种隔离级别，**都不允许脏写的情况发生**，所以没有列入到表格中。
- MySQL与SQL标准不同的一点就是，**MySQL在REPEATABLE READ隔离级别下很大程度地避免了幻读现象**。



## 1.3> MySQL中支持的4种隔离级别

- 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。比如：

- Oracle就只支持READ COMMITTED和SERIALIZABLE这两种隔离级别。
- MySQL支持以上四种，且默认的隔离级别为REPEATABLE READ。

- 设置事务的隔离级别

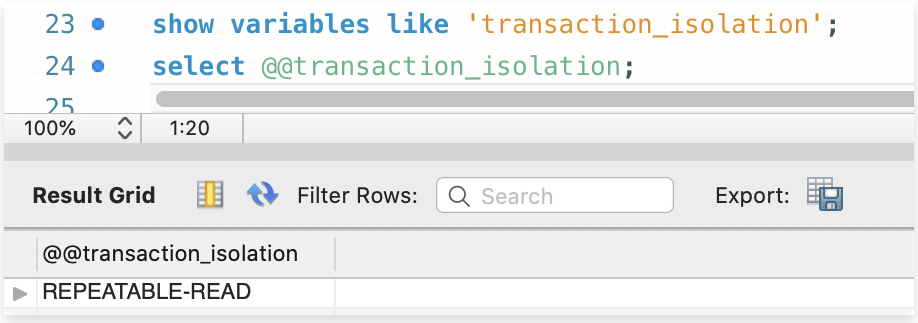
SET [GLOBAL | SESSION | 什么也不加] TRANSACTION ISOLATION LEVEL [READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE];

- GLOBAL
  只对执行完该语句之后**新产生的会话**起作用；
  对**当前已经存在的会话无效**。
- SESSION
  对当前会话所有的**后续事务**有效。
  该语句可以在已经开启的事务中执行，但不会影响当前正在执行的事务。
  如果在事务之间执行，则对后续的事务有效。
- **什么也不加**
  只对当前会话中**下一个即将开启的事务**有效。
  下一个事务执行完毕，**后续事务将恢复到之前的隔离级别**。
  该语句**不能在已经开启的事务中执行**，否则会报错。

- 查看隔离级别

```
show variables like 'transaction_isolation';
select @@transaction_isolation;
```

如下所示：




***注意，transaction_isolation是在MySQL 5.7.20版本中引入的，用来替换tx_isolation。如果大家使用的是之前版本的MySQL，请将其替换为tx_isolation。***

- 查看mysql版本

```
select @@version;
```

- 查看是否自动提交

```
show variables like 'autocommit';
# set autocommit=0;
```

- 需要隔离级别

```
set session transaction isolation level serializable;
set session transaction isolation level read committed;
set session transaction isolation level read uncommitted;
set session transaction isolation level repeatable read;
```



# 二、MVCC原理

## 2.1> 版本链

- **聚簇索引**记录中都包含下面两个必要的**隐藏列**（`row_id`并不是必要的，在创建的表中**有主键**或者**有不允许为NULL的UNIQUE键**时，都不会包含row_id列）。

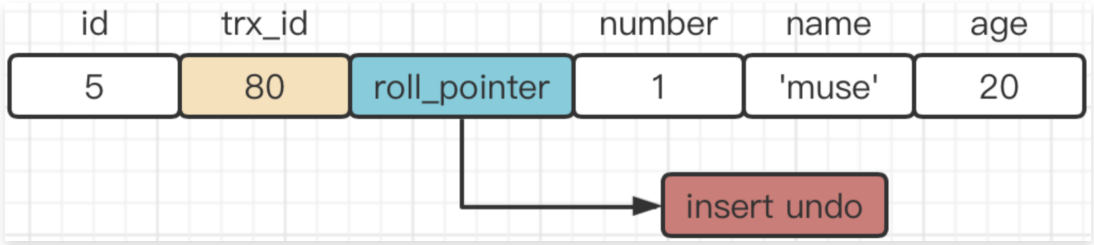
  - trx_id

    一个事务每次对某条聚簇索引记录**进行改动**时（即：`insert`、`delete`、`update`），都会把该事务的事务id赋值给`trx_id`隐藏列。

  - roll_pointer

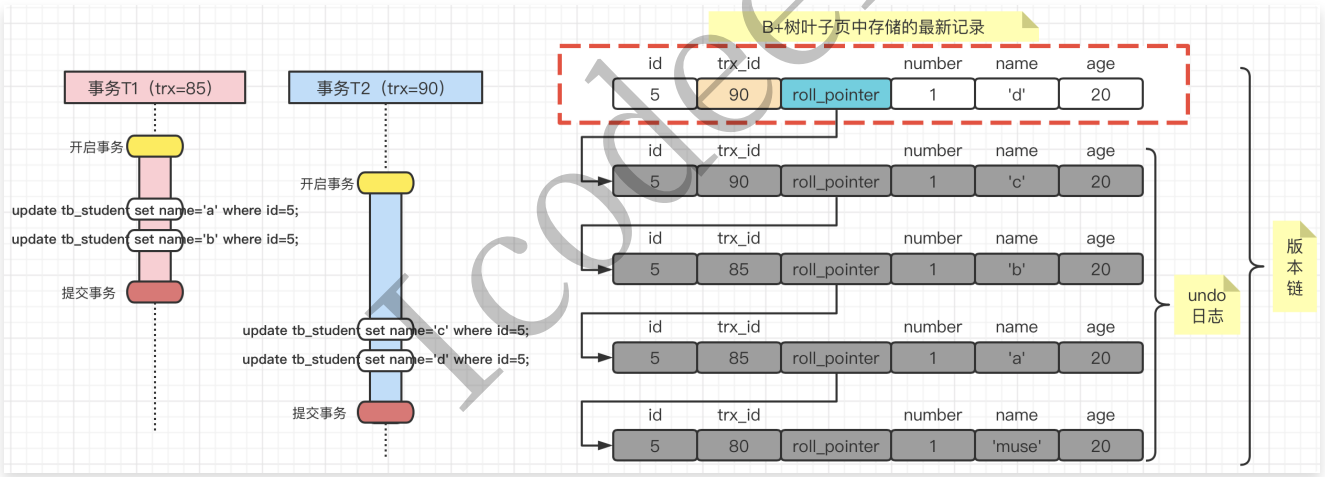
每次对某条聚簇索引记录**进行改动**时，都会把旧的版本写入到undo日志中。这个隐藏列就相当于一个**指针**，可以通过它找到该**记录修改前**的信息。

- 假设执行 `insert into tb_student(id, number, name, age) values(5, 1, 'muse', 20)` ，假设当前执行插入操作的事务id是 `80` ，如下图所示：




【注】实际上insert undo日志只在事务回滚时发生作用。**当事务提交后，该类型的undo日志就没有用了**，它占用的Undo Log Segment也会被系统回收。
虽然真正的insert undo日志占用的存储空间被回收了，**但是roll_pointer的值并不会被清除**。
roll_pointer属性占用7个字节，第一个比特就标记着它指向的undo日志的类型；如果比特值为1，就表示它指向的undo日志属于TRX_UNDO_INSERT大类，也就是该undo日志为insert undo 日志。

- 两个事务T1和T2分别对 `id=5` 的数据进行更新操作，版本链示意图




【注】有一点需要注意，我们在 `UPDATE` 操作产生的undo日志中，**只会记录一些索引列以及被更新的列信息**，并**不会记录所有列的信息**；上图中，记录完整信息，是为了促进理解。

- 什么是版本链

在每次更新该记录后，都会将旧值放到一条undo日志中。随着更新次数的增多，**所有的版本都会被 `roll_pointer` 属性连接成一条链表**，这个链表就称之为**版本链**。

- 版本链的**头节点**就是当前记录的**最新值**。每个版本中还包含生成该版本时对应的**事务id**。
- 什么是MVCC

> 我们利用这个记录的**版本链**和**ReadView**来控制并发事务访问相同记录时的行为，我们把这种机制称之为**多版本并发控制**（`Multi-Version Concurrency Control`）

# 2.2> ReadView

## 2.2.1> 不同隔离级别，访问记录方式

- READ UNCOMMITTED：由于可以读到未提交事务修改过的记录，所以直接读取记录的**最新版本**就好了。
- SERIALIZABLE：InnoDB规定使用**加锁**的方式来访问记录。
- READ COMMITTED & REPEATABLE READ：必须保证读到**已经提交**的事务修改过的记录。

## 2.2.2> ReadView的定义

- ReadView也叫**一致性视图**，用来判断版本链中的**哪个版本是当前事务可见的**。
- ReadView包含4个比较重要的内容：

> - m_ids：在生成ReadView时，当前系统中**活跃的读写事务**的事务id列表。
> - min_trx_id：在生成ReadView时，当前系统中活跃的读写事务中**最小的事务id**；也就是 `m_ids` 中的最小值。
> - max_trx_id：在生成ReadView时，系统应该分配给**下一个事务的事务id**值。
> - creator_trx_id：**生成该ReadView**的事务的事务id。

- **只有在对表中的记录进行改动时（即：insert、delete、update）才会为事务分配唯一的事务id，否则一个事务的事务id值都默认为0。**



## 2.2.3> 如何通过ReadView来判断记录的某个版本是否可见?

*trx_id：表示被访问记录上的事务版本ID——trx_id。*
*`最新/历史版本记录` 的trx_id 与 `Readview` 中的事务id记录进行对比*

- 【case1】如果trx_id == ReadView.creator_trx_id

> 则表明当前事务在访问它**自己修改**过的记录，所以该版本**可以**被当前事务访问。

- 【case2】如果trx_id < ReadView.min_trx_id

> 则表明生成该版本的事务在当前事务**生成ReadView之前**已经提交了，所以该版本**可以**被当前事务访问。

- 【case3】如果trx_id >= ReadView.max_trx_id

> 则表明生成该版本的事务在当前事务**生成ReadView之后**才开启，所以该版本**不可以**被当前事务访问。

- 【case4】如果trx_id in ReadView.m_ids

> 说明创建ReadView时生成该版本的**事务还是活跃的**，该版本**不可以**被访问。

- 【case5】如果trx_id not in ReadView.m_ids

> 说明创建ReadView时生成该版本的**事务已经被提交**，该版本**可以**被访问。

- 如果某个版本的数据对当前事务不可见，那就顺着版本链**找到下一个版本**的数据，并继续执行上面的步骤来判断记录的可见性，以此类推，直到版本链中的最后一个版本。
- `READ COMMITTED` 和 `REPEATABLE READ` 隔离级别之间一个非常大的区别就是——**它们生成ReadView的时机不同！！**
- 前提说明

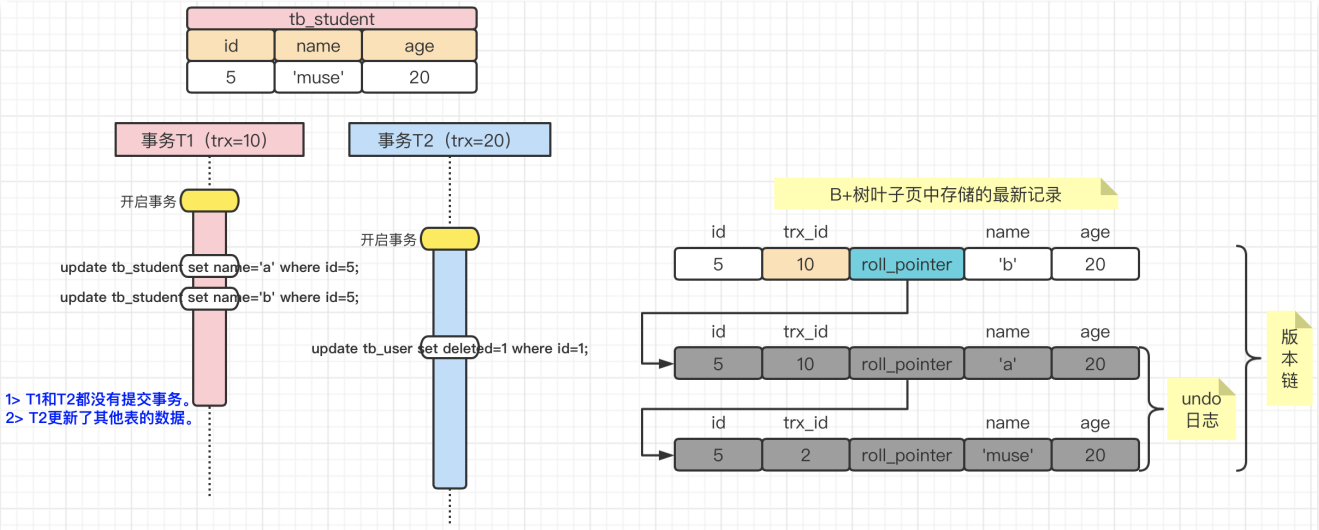
*再次强调，在事务执行过程中，只有在第一次真正修改记录时（比如使用insert、delete、update语句时），才会分配一个唯一的事务id；而且这个事务id是递增的。所以我们才在T2中更新了别的表（tb_user）的记录，目的是为它分配事务id。*

## 2.2.4> READ COMMITED使用ReadView

- `READ COMMITTED` 隔离级别，在一个事务中，**每次 读取数据前 都生成一个ReadView**！！！

### a> 案例1

- 前提条件：**事务T1**针对 `id=5` 的记录执行了两次更新操作；**事务T2**针对 `id=1` 的记录执行了一次更新操作。开启**事务T3**，执行select操作。




> BEGIN; # 开启T3事务
> # 案例1

select name from tb_student where id=5; # 生成ReadView，由于T1和T2都未提交事务，所以查询得到name的值为“ `muse` ”

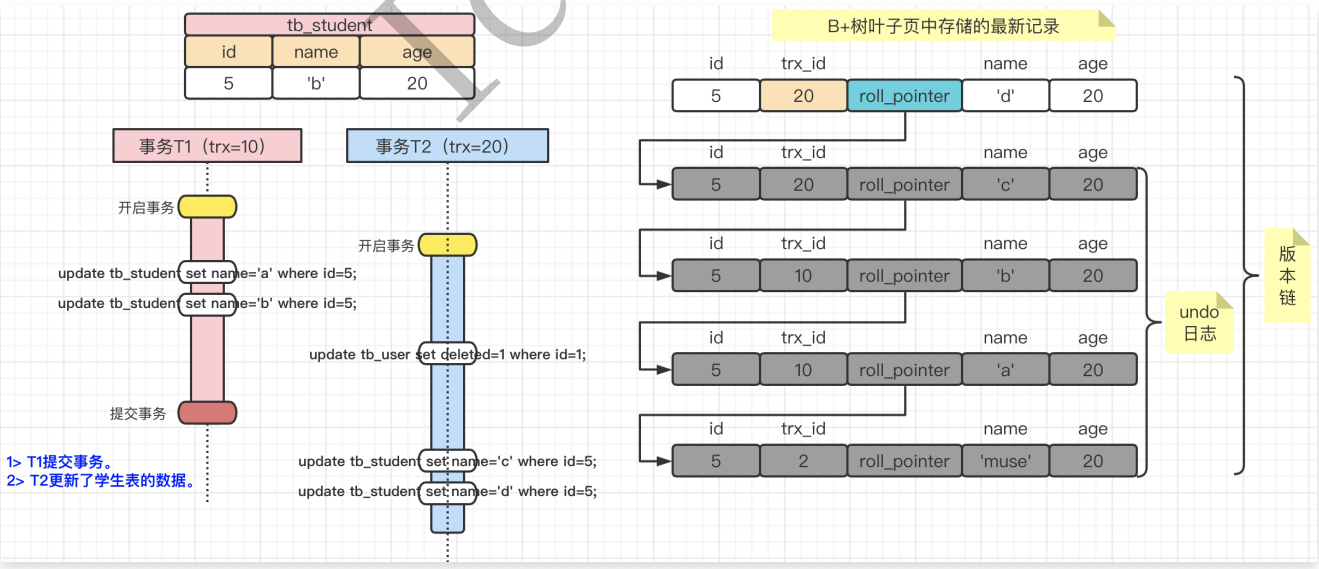
- **步骤1**：此时，再**开启一个新的事物T3**，在这个事务内执行 `select语句` 时，首先生成一个 `ReadView` ：

m_ids=[10, 20]
min_trx_id=10
max_trx_id=21
creator_trx_id=0 （*由于新开启的 事务T3 没有对任何记录进行修改，所以系统不会为它分配唯一的事务id，默认为0*）

- **步骤2**：在最新版本记录中， `name=“b”` 且 `trx_id=10` ，由于 `10` 在ReadView.m_ids列表中，所以不符合可见性要求；所以会根据 `roll_pointer` 跳到下一个版本。
- **步骤3**：由于下一个版本的记录中， `name=“a”` 且 `trx_id=10` ，由于 `10` 在ReadView.m_ids列表中，所以不符合可见性要求；所以会根据 `roll_pointer` 跳到下一个版本。
- **步骤4**：由于下一个版本的记录中， `name=“muse”` 且 `trx_id=2` ，由于2小于ReadView.min_trx_id（即：10），所以符合可见性要求；最后返回给用户name=“muse”的记录。

### b> 案例2

- 前提条件：在上面的案例1之后，T1提交了事务，T2对 `id=5` 的记录进行了两次更新操作，然后我们在T3事务中再次执行 `select` 操作。




BEGIN; # 开启T3事务
# 案例1

> select name from tb_student where id=5; # `生成ReadView`，由于T1和T2都未提交事务，所以查询得到name的值为“`muse`”
> **# ...T1提交事务，T2执行了两次update操作，分别更新name=‘c’和name=‘d’...**
> # 案例2
> select name from tb_student where id=5; # 再次`生成新的ReadView`，由于T1提交事务，所以查询得到name的值为“`b`”

- **步骤1**：第二次执行`select语句`时又会单独生成一个`ReadView`。

> m_ids=[20]（*由于T1已经提交了，所以新生成的ReadView中trx_id=10的活跃事务id就不在m_ids列表中了*）
> min_trx_id=20
> max_trx_id=21
> creator_trx_id=0

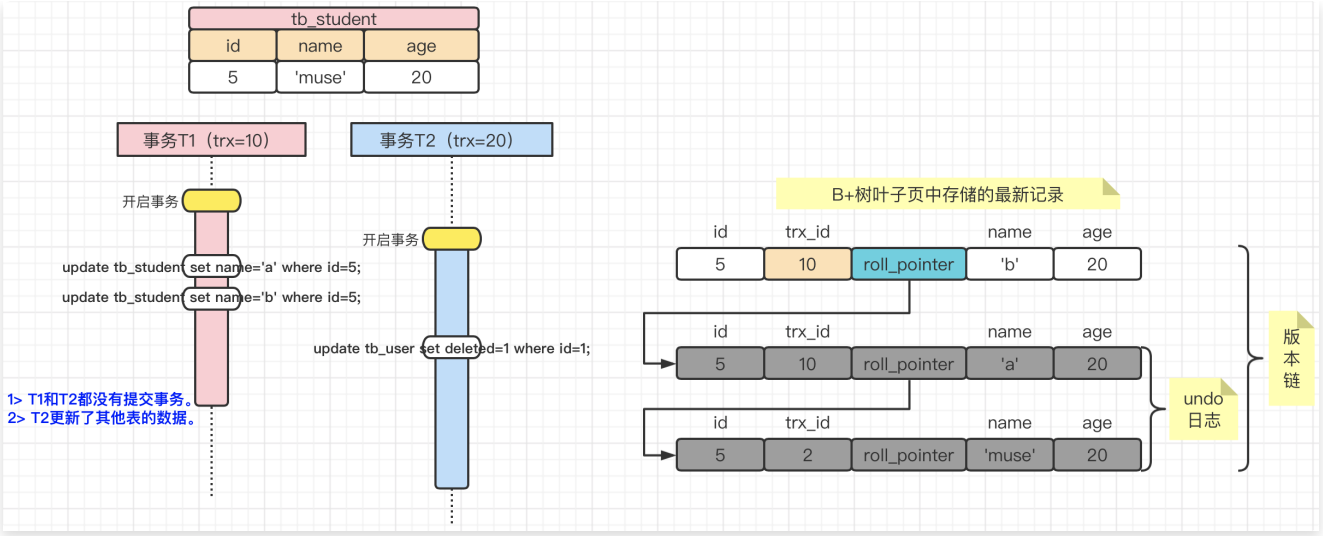
- **步骤2**：在最新版本的记录中，`name=“d”`且`trx_id=20`，由于`20`在ReadView.m_ids列表中，所以不符合可见性要求；然后根据`roll_pointer`跳到下一个版本。
- **步骤3**：由于下一个版本的记录中，`name=“c”`且`trx_id=20`，由于`20`在ReadView.m_ids列表中，所以不符合可见性要求；根据roll_pointer跳到下一个版本。
- **步骤4**：由于下一个版本的记录中，`name=“b”`且`trx_id=10`，由于`10`小于ReadView.min_trx_id(即：20)，所以符合可见性要求；最后返回给用户name=“b”的记录。

## 2.2.5> REPEATABLE READ使用ReadView

- `REPEATABLE READ`隔离级别在一个事务中，只在**第一次读取数据时**生成一个ReadView！！！

### a> 案例1

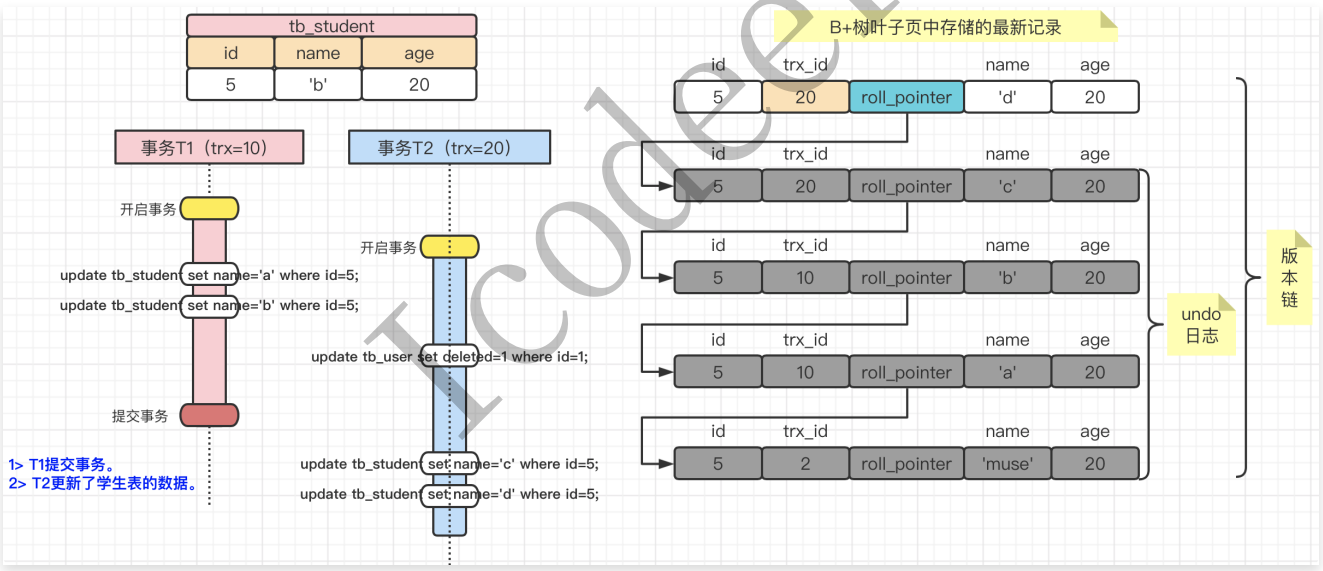
- 前提条件：**事务T1**针对`id=5`的记录执行了两次更新操作；**事务T2**针对`id=1`的记录执行了一次更新操作。开启**事务T3**，执行select操作。




【执行步骤】与READ COMMITTED步骤一样。略

### b> 案例2

- 前提条件：在上面的案例1之后，T1提交了事务，T2对 `id=5` 的记录进行了两次更新操作，然后我们在T3事务中再次执行 `select` 操作。






> BEGIN; # 开启事务
> # 案例1
> select name from tb_student where id=5; # 生成ReadView，由于T1和T2都未提交事务，所以查询得到name的值为“ `muse` ”
> **# ...T1提交事务，T2执行了两次update操作，分别更新name=‘c’和name=‘d’...**
> # 案例2
> select name from tb_student where id=5; # 不生成ReadView了，复用第一次的，虽然T1提交事务，ReadView还是旧的，所以查询得到name的值还为“ `muse` ”

- **步骤1**：第二次执行select语句时，会 `复用` 之前的ReadView。

> m_ids=[10,20]（*即使后续T1提交了，也依然不会生成新的ReadView，trx_id=10这个事务id依然在m_ids中作为活跃事务id*）
> min_trx_id=10
> max_trx_id=21
> creator_trx_id=0

- **步骤2**：在最新版本的记录中，`name="d"` 且 `trx_id=20`，由于 `20` 在**ReadView.m_ids**列表中，所以不符合可见性要求；根据roll_pointer跳到下一个版本。
- **步骤3**：由于在下一个版本的记录中，`name="c"` 且 `trx_id=20`，由于 `20` 在**ReadView.m_ids**列表中，所以不符合可见性要求；根据roll_pointer跳到下一个版本。
- **步骤4**：由于在下一个版本的记录中，`name="b"` 且 `trx_id=10`，由于 `10` 在**ReadView.m_ids**列表中，所以不符合可见性要求；根据roll_pointer跳到下一个版本。
- **步骤5**：由于在下一个版本的记录中，`name="a"` 且 `trx_id=10`，由于 `10` 在**ReadView.m_ids**列表中，所以不符合可见性要求；根据roll_pointer跳到下一个版本。
- **步骤6**：由于在下一个版本的记录中，`name="muse"` 且 `trx_id=2`，由于 `2` 小于**ReadView.min_trx_id**(即：10)，所以符合可见性要求；最后返回给用户name="muse"的记录。

## 2.3> 二级索引与MVCC

- 我们知道，**只有在聚簇索引记录中才有trx_id和roll_pointer隐藏列**。如果某个查询语句使用二级索引来执行查询，该如何判断可见性呢？ 有如下两步

```
BEGIN;  # 开启事务
select name from tb_student where name='muse';
```

- **步骤1**：二级索引页面的Page Header部分有一个名为 `PAGE_MAX_TRX_ID` 的属性，它代表着修改该二级索引页面的 `最大事务id` 是什么。当select语句访问某个二级索引记录时，首先会看一下min_trx_id是否大于该页面的PAGE_MAX_TRX_ID，如果大于，则说明该页面中的所有记录都对该ReadView可见；否则就得执行步骤2，在**回表**之后再判断可见性。
- **步骤2**：利用二级索引记录中的主键值进行回表操作，得到对应的聚簇索引记录后再按照前面讲过的方式找到对该ReadView可见的第一个版本，然后判断该版本中相应的二级索引列（name）的值是否与利用该二级索引查询时的值（"muse"）相同。

## 2.4> MVCC小结

- 所谓的MVCC指的就是在使用 `READ COMMITTED` 和 `REPEATABLE READ` 这两种隔离级别的事务执行普通的 `SELECT操作` 时，访问记录的 `版本链` 的过程。
- 这样可以使不同事务的 `读-写` 、 `写-读` 操作并发执行，从而提升性能。

- READ COMMITTED和REPEATABLE READ这两种隔离级别有一个很大的不同点——就是**生成ReadView的时间不同**。
- READ COMMITTED在**每一次**进行普通SELECT操作前都会生成一个ReadView。
- REPEATABLE READ只会在**第一次**进行普通SELECT操作前生成一个ReadView。

---

吾尝终日而思矣，不如须臾之所学也；
吾尝跂而望矣，不如登高之博见也。
登高而招，臂非加长也，而见者远；
顺风而呼，声非加疾也，而闻者彰。
假舆马者，非利足也，而致千里；
假舟楫者，非能水也，而绝江河。
君子生非异也，善假于物也。
------ 摘自《劝学》

愿本文可以成为大家的“山”、“风”、“马”、“舟”，助大家在技术之路上乘风破浪，大展宏图~~

同时，也欢迎大家关注我的公众号“**爪哇缪斯**” ~\(^o^)/~ 「干货分享，每天更新」