

MySQL事务篇

一、一条Insert语句

为了故事的顺利发展，我们需要创建一个表：

```
1 CREATE TABLE t (  
2     id INT PRIMARY KEY,  
3     c VARCHAR(100)  
4 ) Engine=InnoDB CHARSET=utf8;
```

然后向这个表里插入一条数据：

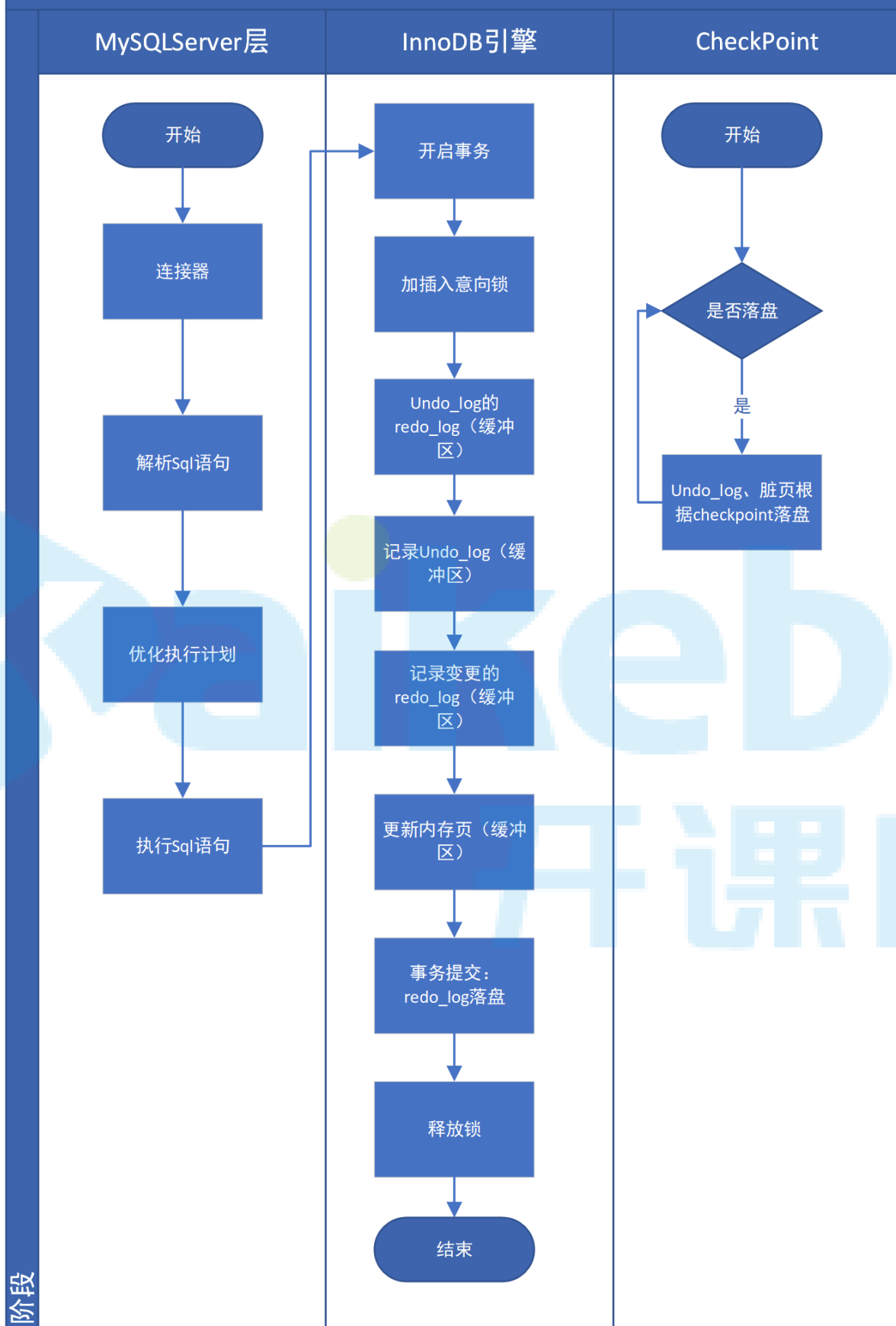
```
1 INSERT INTO t VALUES(1, '刘备');
```

现在表里的数据就是这样的：

```
1 mysql> SELECT * FROM t;  
2 +----+-----+  
3 | id | c      |  
4 +----+-----+  
5 | 1  | 刘备   |  
6 +----+-----+  
7 1 row in set (0.01 sec)
```

二、执行流程

Insert语句执行流程

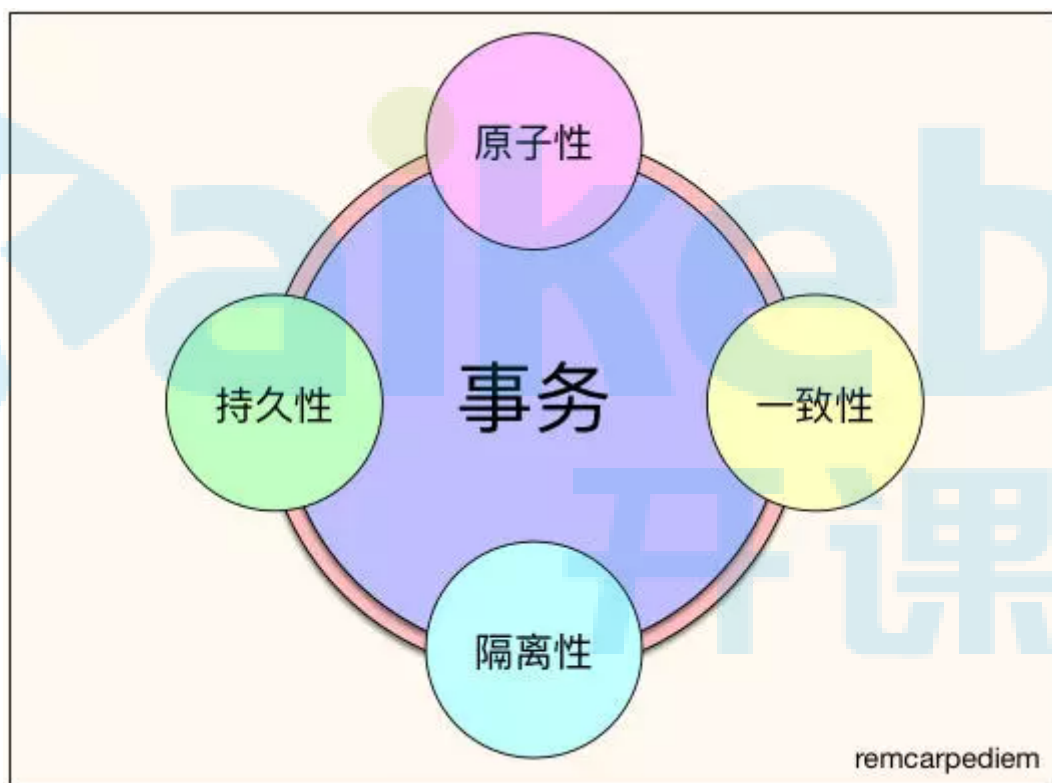


二、事务介绍

1、事务概述

MySQL 是一个服务器 / 客户端架构的软件，对于同一个服务器来说，可以有若干个客户端与之连接，每个客户端与服务器连接上之后，就可以称之为一个会话（Session）。我们可以同时在不同的会话里输入各种语句，这些语句可以作为事务的一部分进行处理。不同的会话可以同时发送请求，也就是说服务器可能同时在处理多个事务，这样子就会导致不同的事务可能同时访问到相同的记录。我们前边说过事务有一个特性称之为 **隔离性**，理论上在某个事务对某个数据进行访问时，其他事务应该进行排队，当该事务提交之后，其他事务才可以继续访问这个数据。但是这样子的话对性能影响太大，所以设计数据库的大叔提出了各种 **隔离级别**，来最大限度的提升系统并发处理事务的能力，但是这也是以牺牲一定的 **隔离性** 来达到的。

事务是数据库最为重要的机制之一，凡是使用过数据库的人，都了解数据库的事务机制，也对ACID四个基本特性如数家珍。但是聊起事务或者ACID的底层实现原理，往往言之不详，不明所以。在[MySQL中的事务是由存储引擎实现的](#)，而且支持事务的存储引擎不多，我们主要讲解InnoDB存储引擎中的事务。所以，今天我们就一起来分析和探讨InnoDB的事务机制，希望能建立起对事务底层实现原理的具体了解。



事务的四大特性

数据库事务具有ACID四大特性。ACID是以下4个词的缩写：

- [原子性\(atomicity\)](#)：事务最小工作单元，要么全成功，要么全失败。
- [一致性\(consistency\)](#)：事务开始和结束后，数据库的完整性不会被破坏。
- [隔离性\(isolation\)](#)：不同事务之间互不影响，四种隔离级别为RU（读未提交）、RC（读已提交）、RR（可重复读）、SERIALIZABLE（串行化）。
- [持久性\(durability\)](#)：事务提交后，对数据的修改是永久性的，即使系统故障也不会丢失。

2、隔离级别

1) 未提交读 (READ UNCOMMITTED/RU)

脏读：一个事务读取到另一个事务**未提交**的数据。

如果一个事务读到了另一个未提交事务修改过的数据，那么这种隔离级别就称之为**未提交读**（英文名：READ UNCOMMITTED），示意图如下：

READ UNCOMMITTED 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE t SET c = '关羽' WHERE id = 1;
④	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'关羽')	

如上图，Session A 和 Session B 各开启了一个事务，Session B 中的事务先将 id 为 1 的记录列 c 更新为 '关羽'，然后 Session A 中的事务再去查询这条 id 为 1 的记录，那么在未提交读的隔离级别下，查询结果就是 '关羽'，也就是说某个事务读到了另一个未提交事务修改过的记录。但是如果 Session B 中的事务稍后进行了回滚，那么 Session A 中的事务相当于读到了一个不存在的数据，这种现象就称之为**脏读**，就像这个样子：

READ UNCOMMITTED 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE t SET c = '关羽' WHERE id = 1;
④	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'关羽')	
⑤		ROLLBACK;

脏读违背了现实世界的业务含义，所以这种 READ UNCOMMITTED 算是十分不安全的一种隔离级别。

2) 已提交读 (READ COMMITTED/RC)

不可重复读：一个事务因读取到另一个事务**已提交的update**。导致对**同一条记录**读取两次以上的结果不一致。

如果一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那么这种隔离级别就称之为已提交读（英文名：READ COMMITTED），如图所示：

READ COMMITTED 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE t SET c = '关羽' WHERE id = 1;
④	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	
⑤		COMMIT;
⑥	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'关羽')	

从图中可以看到，第4步时，由于 Session B 中的事务尚未提交，所以 Session A 中的事务查询得到的结果只是 '刘备'，而第6步时，由于 Session B 中的事务已经提交，所以 Session A 中的事务查询得到的结果就是 '关羽' 了。

对于某个处在在已提交读隔离级别下的事务来说，只要其他事务修改了某个数据的值，并且之后提交了，那么该事务就会读到该数据的最新值，比方说：

READ COMMITTED 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	
③		UPDATE t SET c = '关羽' WHERE id = 1; (隐式提交)
④	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'关羽')	
⑤		UPDATE t SET c = '张飞' WHERE id = 1; (隐式提交)
⑥	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'张飞')	

我们在 Session B 中提交了几个隐式事务，这些事务都修改了 id 为 1 的记录的列c的值，每次事务提交之后，Session A 中的事务都可以查看到最新的值。这种现象也被称之为不可重复读。

3) 可重复读 (REPEATABLE READ/RR)

幻读：一个事务因读取到另一个事务已提交的insert数据。导致对同一张表读取两次以上的结果不一致。

在一些业务场景中，一个事务只能读到另一个已经提交的事务修改过的数据，但是第一次读过某条记录后，即使其他事务修改了该记录的值并且提交，该事务之后再读该条记录时，读到的仍是第一次读到的值，而不是每次都读到不同的数据。那么这种隔离级别就称之为可重复读（英文名：REPEATABLE READ），如图所示：

REPEATABLE READ 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	
③		UPDATE t SET c = '关羽' WHERE id = 1; (隐式提交)
④	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	
⑤		UPDATE t SET c = '张飞' WHERE id = 1; (隐式提交)
⑥	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	

从图中可以看出，Session A 中的事务在第一次读取 id 为 1 的记录时，列 c 的值为 '刘备'，之后虽然 Session B 中隐式提交了多个事务，每个事务都修改了这条记录，但是 Session A 中的事务读到的列 c 的值仍为 '刘备'，与第一次读取的值是相同的。

4) 串行化 (SERIALIZABLE)

以上3种隔离级别都允许对同一条记录进行读-读、读-写、写-读的并发操作，如果我们不允许读-写、写-读的并发操作，可以使用SERIALIZABLE隔离级别，示意图如下：

SERIALIZABLE 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE t SET c = '关羽' WHERE id = 1;
④	SELECT * FROM t WHERE id = 1; (等待中...)	
⑤		COMMIT;
⑥	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'关羽')	

如图所示，当 Session B 中的事务更新了 id 为 1 的记录后，之后 Session A 中的事务再去访问这条记录时就被卡住了，直到 Session B 中的事务提交之后，Session A 中的事务才可以获取到查询结果。

备注：设置当前会话的事务隔离级别

```
1 //查看当前事务级别：
2 SELECT @@tx_isolation;
3 //设置read uncommitted级别：
4 set session transaction isolation level read uncommitted;
5
6 //设置read committed级别：
7 set session transaction isolation level read committed;
8
9 //设置repeatable read级别：
10 set session transaction isolation level repeatable read;
11
12 //设置serializable级别：
13 set session transaction isolation level serializable;
```

三、事务和MVCC底层原理详解

1、思考：丢失更新

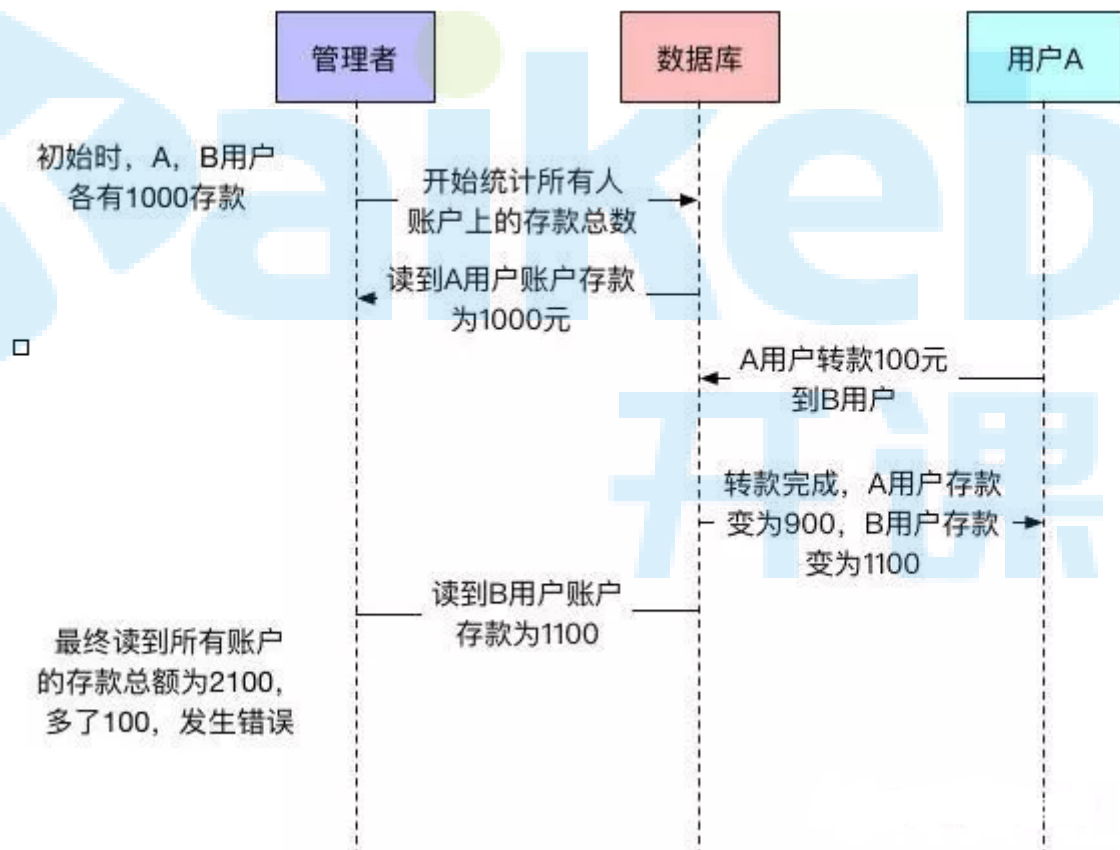
两个事务针对同一数据都发生修改操作时，会存在丢失更新问题。

时间	取款事务A	转账事务B
T1	开始事务	
T2		开始事务
T3	查询账户余额为1000元	
T4		查询账户余额为1000元
T5		汇入100元把余额改为1100元
T6		提交事务
T7	取出100元把余额改为900元	
T8	撤销事务	
T9	余额恢复为1000元（丢失更新）	

时间	转账事务A	取款事务B
T1		开始事务
T2	开始事务	
T3		查询账户余额为1000元
T4	查询账户余额为1000元	
T5		取出100元把余额改为900元
T6		提交事务
T7	汇入100元	
T8	提交事务	
T9	把余额改为1100元（丢失更新）	

再看一个例子：管理者要查询所有用户的存款总额，假设除了用户A和用户B之外，其他用户的存款总额都为0，A、B用户各有存款1000，所以所有用户的存款总额为2000。但是在查询过程中，用户A会向用户B进行转账操作。转账操作和查询总额操作的时序图如下图所示。

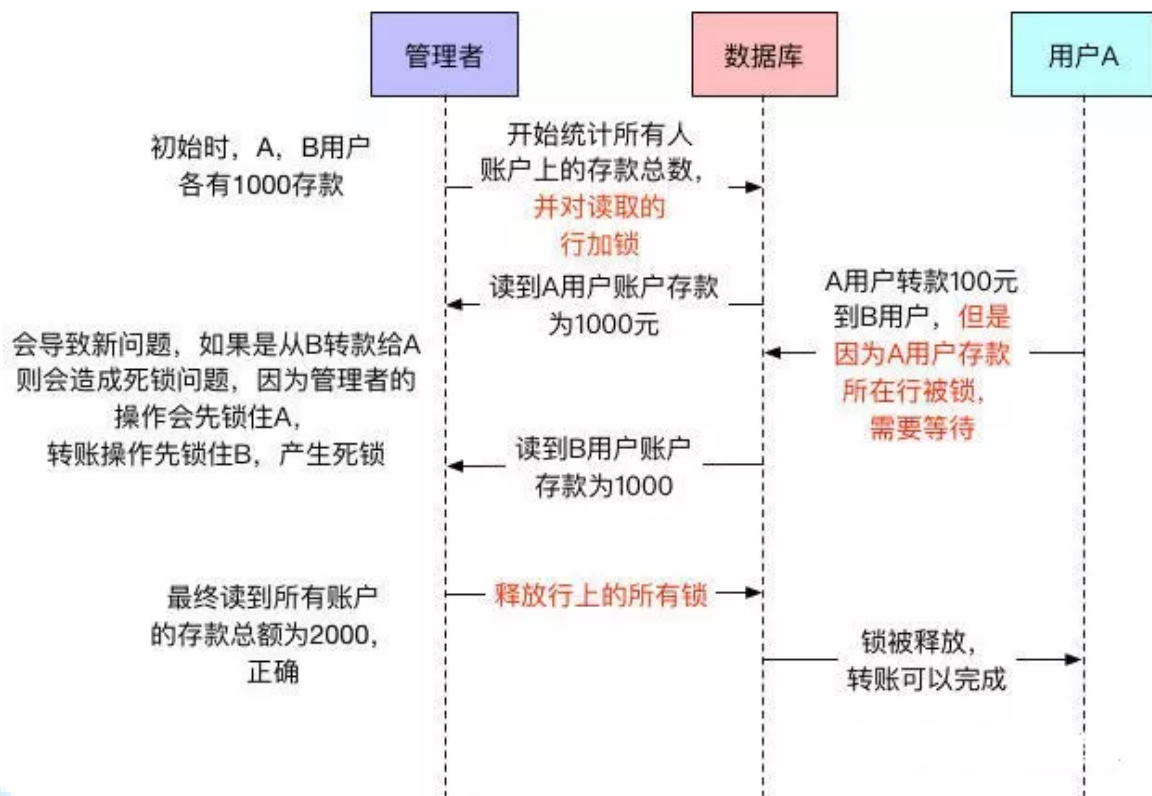
转账和查询的时序图：



2、解决方案1：LBCC

使用LBCC（LBCC，基于锁的并发控制，英文全称Lock Based Concurrency Control）可以解决上述的问题。查询总额事务会对读取的行加锁，等到操作结束后再释放所有行上的锁。因为用户A的存款被锁，导致转账操作被阻塞，直到查询总额事务提交并将所有锁都释放。

使用锁机制：

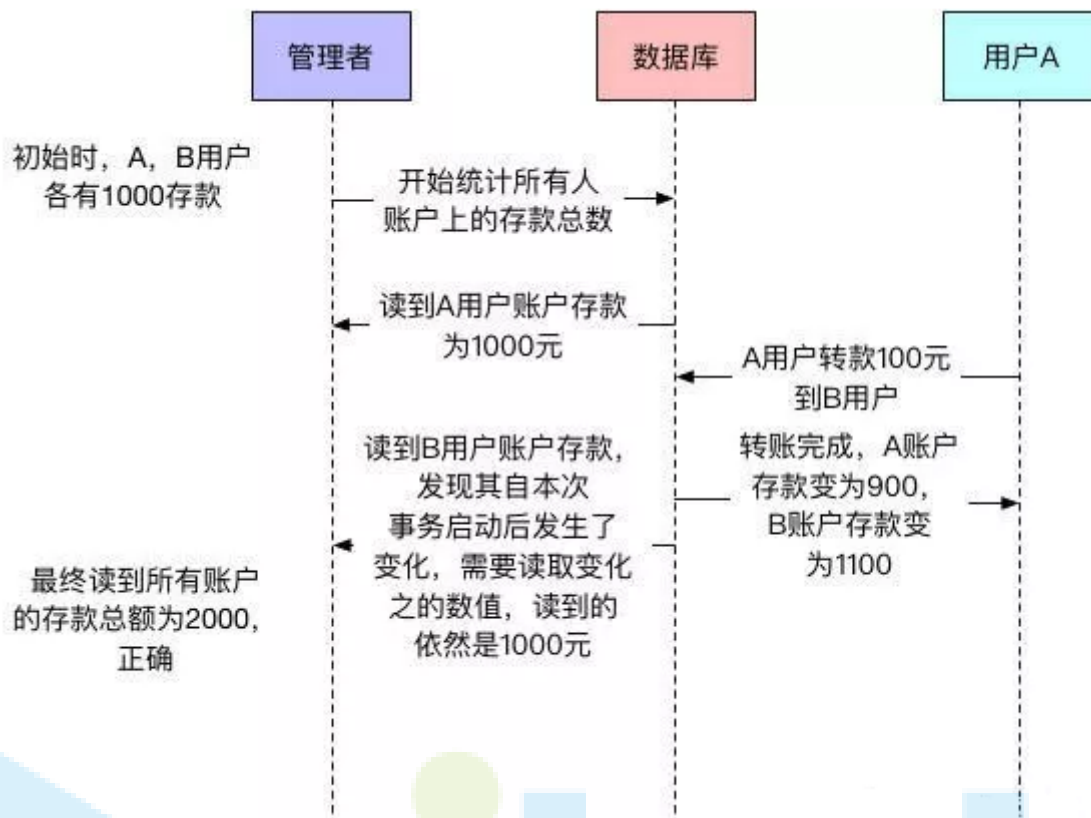


这种方案比较简单粗暴, 就是一个事务去读取一条数据的时候, 就上锁, 不允许其他事务来操作。MySQL加锁之后就是**当前读**。假如当前事务只是加共享锁, 那么其他事务就不能有排他锁, 也就是不能修改数据; 而假如当前事务需要加排他锁, 那么其他事务就不能持有任何锁。总而言之, 能加锁成功, 就确保了除了当前事务之外, 其他事务不会对当前数据产生影响, 所以自然而然的, 当前事务读取到的数据就只能是最新的, 而不会是快照数据(后文MVCC会解释快照读概念)。

3、解决方案2: MVCC

当然使用MVCC (MVCC,多版本的并发控制, 英文全称: Multi Version Concurrency Control) 机制可以解决这个问题。查询总额事务先读取了用户A的账户存款, 然后转账事务会修改用户A和用户B账户存款, 查询总额事务读取用户B存款时不会读取转账事务修改后的数据, 而是读取本事务开始时的数据副本(在REPEATABLE READ隔离等级下)。

使用MVCC机制（RR隔离级别下的演示情况）：



MVCC使得数据库读不会对数据加锁，普通的SELECT请求不会加锁，提高了数据库的并发处理能力。借助MVCC，数据库可以实现READ COMMITTED，REPEATABLE READ等隔离级别，用户可以查看当前数据的前一个或者前几个历史版本，保证了ACID中的I特性（隔离性）。

4、InnoDB的MVCC实现

我们首先来看一下wiki上对MVCC的定义：

Multiversion concurrency control (MCC or MVCC), is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.

由定义可知，[MVCC是用于数据库提供并发访问控制的并发控制技术](#)。与MVCC相对的是基于锁的并发控制，[Lock-Based Concurrency Control \(LBCC\)](#)。MVCC最大的好处，相信也是耳熟能详：[读不加锁，读写不冲突](#)。在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的RDBMS，都支持了MVCC。

多版本并发控制仅仅是一种技术概念，并没有统一的实现标准，[其核心理念就是数据快照，不同的事务访问不同版本的数据快照，从而实现不同的事务隔离级别](#)。虽然字面上是说具有多个版本的数据快照，但这并不意味着数据库必须拷贝数据，保存多份数据文件，这样会浪费大量的存储空间。InnoDB通过事务的undo日志巧妙地实现了多版本的数据快照。

MVCC 在mysql 中的实现依赖的是 **undo log** 与 **read view** 。

InnoDB的MVCC是通过在每行记录后面保存两个隐藏的列来实现的。这两个列，**一个保存了行的事务ID，一个保存了行的回滚指针**。每开始一个新的事务，都会自动递增产生一个新的事务id。事务开始时刻的会把事务id放到当前事务影响的行事务id中，当查询时需要用当前事务id和每行记录的事务id进行比较。

MVCC只在REPEATABLE READ和READ COMMITTED两个隔离级别下工作。其他两个隔离级别都和MVCC不兼容，因为READ UNCOMMITTED总是读取最新的数据行，而不是符合当前事务版本的数据行。而SERIALIZABLE则会对所有读取的行都加锁。

1) undo log

根据行为的不同，undo log分为两种：insert undo log和update undo log

- insert undo log:

- 1 是在 insert 操作中产生的 undo log。
- 2 因为 insert 操作的记录只对事务本身可见，对于其它事务此记录是不可见的，所以 insert undo log 可以在事务提交后直接删除而不需要进行 purge 操作。

为了更好的支持并发，InnoDB的多版本一致性读是采用了基于回滚段的方式。另外，对于更新和删除操作，InnoDB并不是真正的删除原来的记录，而是设置记录的delete mark为1。因此为了解决数据Page和Undo Log膨胀的问题，需要引入purge机制进行回收。Undo log保存了记录修改前的镜像。在InnoDB存储引擎中，undo log分为：

- insert undo log
- update undo log

insert undo log是指在insert操作中产生的undo log。由于insert操作的记录，只是对本事务可见，其他事务不可见，所以undo log可以在事务提交后直接删除，而不需要purge操作。

update undo log是指在delete和update操作中产生的undo log。该undo log会被后续用于MVCC当中，因此不能提交的时候删除。提交后会放入undo log的链表，等待purge线程进行最后的删除。

如下图所示（初始状态）：

事务1: INSERT INTO user(id, name, age, address)
VALUES (10, 'Tom', 23, 'NanJing')

	事务ID DB_TRX_ID	回滚指针 DB_ROLL_PT				
RowID	1	NULL	10	'Tom'	23	'NanJing'

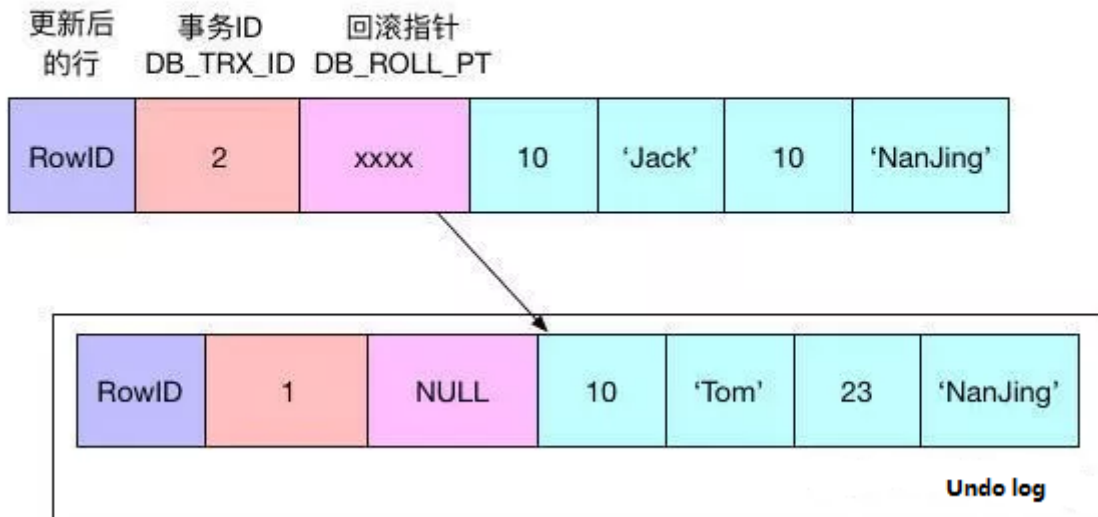
- update undo log :

- 1 是 update 或 delete 操作中产生的 undo log。
- 2 因为会对已经存在的记录产生影响，为了提供 MVCC机制，因此 update undo log 不能在事务提交时就进行删除，而是将事务提交时放到入 history list 上，等待 purge 线程进行最后的删除操作。

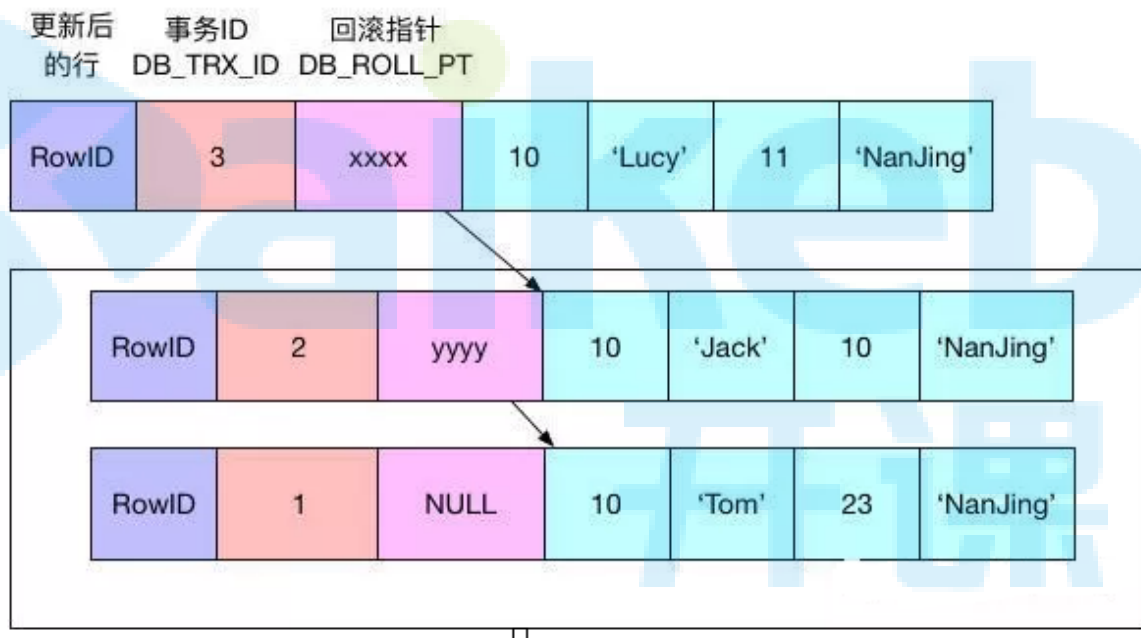
如下图所示（第一次修改）：

当事务2使用UPDATE语句修改该行数据时，会首先使用排他锁锁定该行，将该行当前的值复制到undo log中，然后再真正地修改当前行的值，最后填写事务ID，使用回滚指针指向undo log中修改前的行。

事务2: UPDATE user SET name='Jack', age=10 WHERE id = 10



当事务3进行修改与事务2的处理过程类似，如下图所示（第二次修改）：



为了保证事务并发操作时，在写各自的undo log时不产生冲突，InnoDB采用回滚段的方式来维护undo log的并发写入和持久化。回滚段实际上是一种 Undo 文件组织方式。

2) ReadView

对于使用 READ UNCOMMITTED 隔离级别的事务来说，直接读取记录的最新版本就好了。对于使用 SERIALIZABLE 隔离级别的事务来说，使用加锁的方式来访问记录。对于使用 READ COMMITTED 和 REPEATABLE READ 隔离级别的事务来说，就需要用到我们上边所说的 版本链 了。核心问题就是需要判断一下版本链中的哪个版本是当前事务可见的。所以设计 InnoDB 的设计者提出了一个ReadView的概念，这个 ReadView 中主要包含当前系统中还有哪些活跃的读写事务，把它们的事务id放到一个列表中，我们把这个列表命名为 **m_ids**，并确定三个变量的值：

m_up_limit_id: m_ids事务列表中的最小事务id，如果当前列表为空那么就等于m_low_limit_id。事务id的下限。

m_low_limit_id: 系统中将要产生的下一个事务id的值。事务id的上限。

m_creator_trx_id: 当前事务id，m_ids中不包含当前事务id。

这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本（[版本链中的版本](#)）是否可见：

- 如果被访问版本的 `trx_id` 属性值小于 `m_up_limit_id`，表明生成该版本的事务在生成 `ReadView` 前已经提交，所以该版本**可以被**当前事务访问。
- 如果被访问版本的 `trx_id` 属性值等于 `m_creator_trx_id` 即当前事务id，**可以被**访问。
- 如果被访问版本的 `trx_id` 属性值大于等于 `m_low_limit_id`，在生成 `ReadView` 后才生成，所以该版本**不可以**被当前事务访问。
- 如果被访问版本的 `trx_id` 属性值在 `m_up_limit_id` 和 `m_low_limit_id` 之间，那就需要判断一下 `trx_id` 属性值是不是在 `m_ids` 列表中。
 - 如果在，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；
 - 如果不在，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的话，那就顺着版本链找到下一个版本的数据，继续按照上边的步骤判断可见性，依此类推，直到版本链中的最后一个版本，如果最后一个版本也不可见的话，那么就意味着该条记录对该事务不可见，查询结果就不包含该记录。

在MySQL中，`READ COMMITTED`和`REPEATABLE READ`隔离级别的的一个非常大的区别就是它们生成`ReadView`的时机不同，我们来看一下。

1.READ COMMITTED

每次读取数据前都生成一个`ReadView`

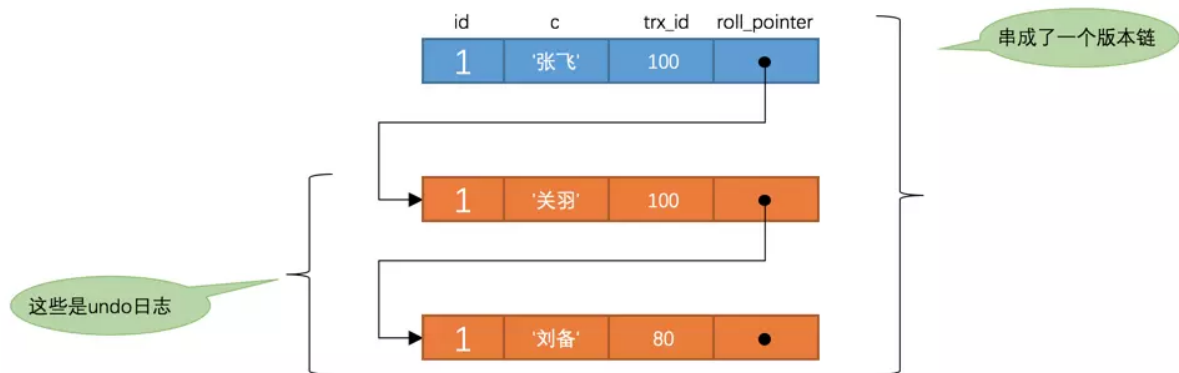
比方说现在系统里有两个id分别为100、200的事务在执行：

```
1 # Transaction 100
2 BEGIN;
3
4 UPDATE t SET c = '关羽' WHERE id = 1;
5
6 UPDATE t SET c = '张飞' WHERE id = 1;
```

```
1 # Transaction 200
2 BEGIN;
3
4 # 更新了一些别的表的记录
5 ...
```

小贴士：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），才会被分配一个单独的事务id，这个事务id是递增的。

此刻，表`t`中id为1的记录得到的版本链表如下所示：



假设现在有一个使用 `READ COMMITTED` 隔离级别的事务开始执行：

```
1 # 使用READ COMMITTED隔离级别的事务
2 BEGIN;
3
4 # SELECT1: Transaction 100、200未提交
5 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
```

这个 `SELECT1` 的执行过程如下：

- 在执行 `SELECT` 语句时会先生成一个 `ReadView`，`ReadView` 的 `m_ids` 列表的内容就是 `[100, 200]`。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 `c` 的内容是 '张飞'，该版本的 `trx_id` 值为 100，在 `m_ids` 列表内，所以不符合可见性要求，根据 `roll_pointer` 跳到下一个版本。
- 下一个版本的列 `c` 的内容是 '关羽'，该版本的 `trx_id` 值也为 100，也在 `m_ids` 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 `c` 的内容是 '刘备'，该版本的 `trx_id` 值为 80，小于 `m_ids` 列表中最小的事务 id 100，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 `c` 为 '刘备' 的记录。

之后，我们把事务id为 100 的事务提交一下，就像这样：

```
1 # Transaction 100
2 BEGIN;
3
4 UPDATE t SET c = '关羽' WHERE id = 1;
5
6 UPDATE t SET c = '张飞' WHERE id = 1;
7
8 COMMIT;
```

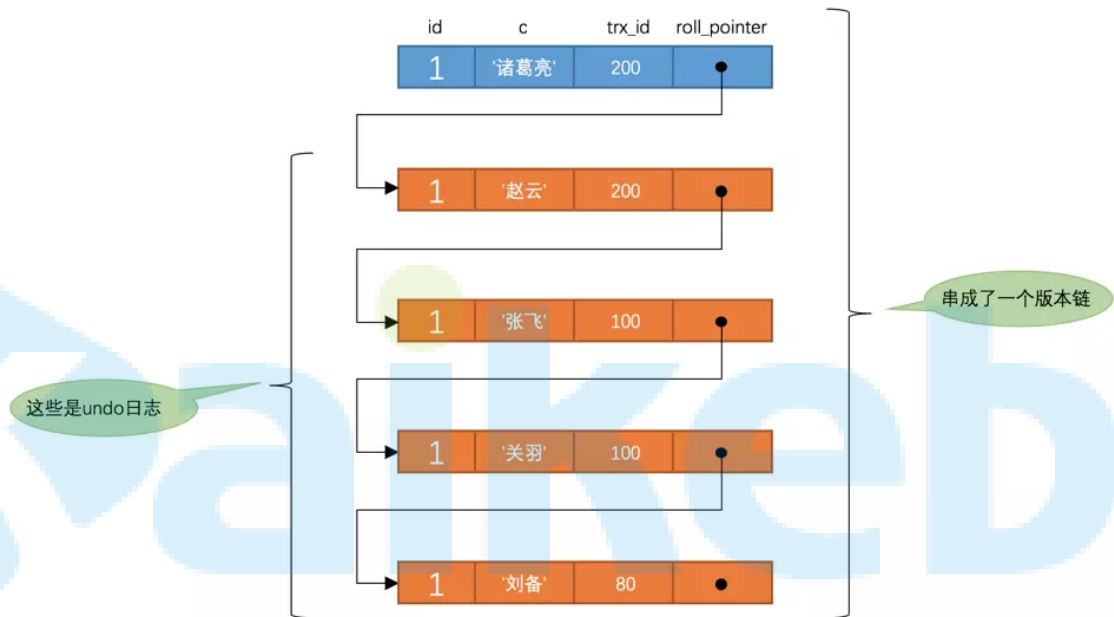
然后再到事务id为 200 的事务中更新一下表 `t` 中 `id` 为 1 的记录：


```

1 # Transaction 200
2 BEGIN;
3
4 # 更新了一些别的表的记录
5 ...
6
7 UPDATE t SET c = '赵云' WHERE id = 1;
8
9 UPDATE t SET c = '诸葛亮' WHERE id = 1;

```

此刻，表 `t` 中 `id` 为 1 的记录的版本链就长这样：



然后再到刚才使用 `READ COMMITTED` 隔离级别的事务中继续查找这个 `id` 为 1 的记录，如下：

```

1 # 使用READ COMMITTED隔离级别的事务
2 BEGIN;
3
4 # SELECT1: Transaction 100、200均未提交
5 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
6
7 # SELECT2: Transaction 100提交, Transaction 200未提交
8 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'张飞'

```

这个 `SELECT2` 的执行过程如下：

- 在执行 `SELECT` 语句时会先生成一个 `ReadView`，`ReadView` 的 `m_ids` 列表的内容就是 `[200]`（事务 `id` 为 100 的那个事务已经提交了，所以生成快照时就没有它了）。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 `c` 的内容是 '诸葛亮'，该版本的 `trx_id` 值为 200，在 `m_ids` 列表内，所以不符合可见性要求，根据 `roll_pointer` 跳到下一个版本。
- 下一个版本的列 `c` 的内容是 '赵云'，该版本的 `trx_id` 值为 200，也在 `m_ids` 列表内，所以也不符合要求，继续跳到下一个版本。

- 下一个版本的列 c 的内容是 '张飞'，该版本的 trx_id 值为 100，比 m_ids 列表中最小的事务 id 200 还要小，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 c 为 '张飞' 的记录。

以此类推，如果之后事务 id 为 200 的记录也提交了，再此在使用 READ COMMITTED 隔离级别的事务中查询表 t 中 id 值为 1 的记录时，得到的结果就是 '诸葛亮' 了，具体流程我们就不分析了。

总结一下就是：[使用READ COMMITTED隔离级别的事务在每次查询开始时都会生成一个独立的ReadView。](#)

2.REPEATABLE READ

在事务开始后第一次读取数据时生成一个ReadView

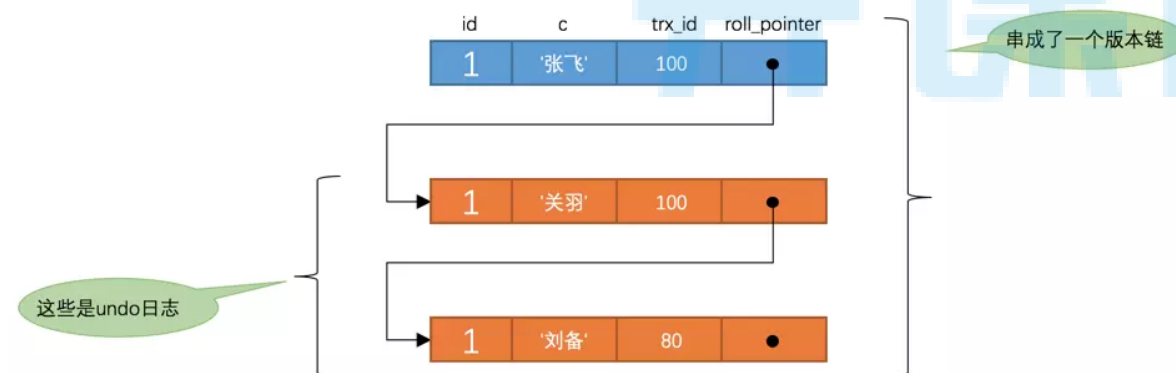
对于使用 REPEATABLE READ 隔离级别的事务来说，只会在第一次执行查询语句时生成一个 ReadView，之后的查询就不会重复生成了。我们还是用例子看一下是什么效果。

比方说现在系统里有两个 id 分别为 100、200 的事务在执行：

```
1 # Transaction 100
2 BEGIN;
3
4 UPDATE t SET c = '关羽' WHERE id = 1;
5
6 UPDATE t SET c = '张飞' WHERE id = 1;
```

```
1 # Transaction 200
2 BEGIN;
3
4 # 更新了一些别的表的记录
5 ...
```

此刻，表 t 中 id 为 1 的记录得到的版本链表如下所示：



假设现在有一个使用 REPEATABLE READ 隔离级别的事务开始执行：

```
1 # 使用REPEATABLE READ隔离级别的事务
2 BEGIN;
3
4 # SELECT1: Transaction 100、200未提交
5 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
```


这个 SELECT 的执行过程如下：

- 在执行 SELECT 语句时会先生成一个 ReadView，ReadView 的 m_ids 列表的内容就是 [100, 200]。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 c 的内容是 '张飞'，该版本的 trx_id 值为 100，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 c 的内容是 '关羽'，该版本的 trx_id 值也为 100，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 c 的内容是 '刘备'，该版本的 trx_id 值为 80，小于 m_ids 列表中最小的事务 id 100，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 c 为 '刘备' 的记录。

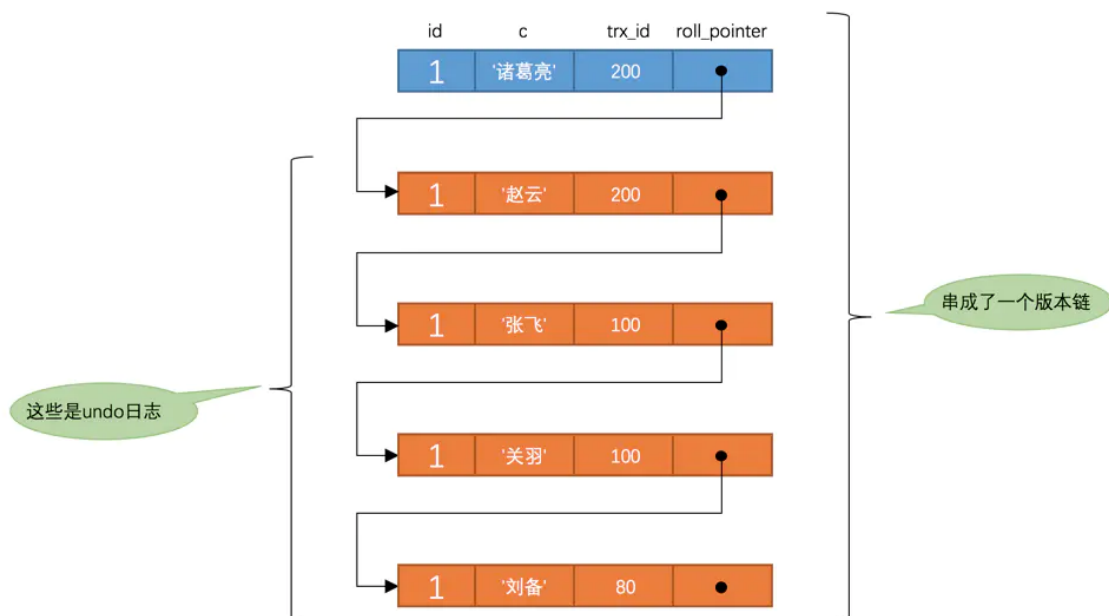
之后，我们把事务 id 为 100 的事务提交一下，就像这样：

```
1 # Transaction 100
2 BEGIN;
3
4 UPDATE t SET c = '关羽' WHERE id = 1;
5
6 UPDATE t SET c = '张飞' WHERE id = 1;
7
8 COMMIT;
```

然后再到事务 id 为 200 的事务中更新一下表 t 中 id 为 1 的记录：

```
1 # Transaction 200
2 BEGIN;
3
4 # 更新了一些别的表的记录
5 ...
6
7 UPDATE t SET c = '赵云' WHERE id = 1;
8
9 UPDATE t SET c = '诸葛亮' WHERE id = 1;
```

此刻，表 t 中 id 为 1 的记录的版本链就长这样：



然后再到刚才使用 REPEATABLE READ 隔离级别的事务中继续查找这个id为 1 的记录，如下：

```

1 # 使用REPEATABLE READ隔离级别的事务
2 BEGIN;
3
4 # SELECT1: Transaction 100、200均未提交
5 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
6
7 # SELECT2: Transaction 100提交, Transaction 200未提交
8 SELECT * FROM t WHERE id = 1; # 得到的列c的值仍为'刘备'

```

这个 SELECT2 的执行过程如下：

- 因为之前已经生成过 ReadView 了，所以此时直接复用之前的 ReadView，之前的 ReadView 中的 m_ids 列表就是 [100, 200]。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 c 的内容是 '诸葛亮'，该版本的 trx_id 值为 200，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 c 的内容是 '赵云'，该版本的 trx_id 值为 200，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 c 的内容是 '张飞'，该版本的 trx_id 值为 100，而 m_ids 列表中是包含值为 100 的事务id的，所以该版本也不符合要求，同理下一个列 c 的内容是 '关羽' 的版本也不符合要求。继续跳到下一个版本。
- 下一个版本的列 c 的内容是 '刘备'，该版本的 trx_id 值为 80，80 小于 m_ids 列表中最小的事务id 100，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 c 为 '刘备' 的记录。

也就是说两次 SELECT 查询得到的结果是重复的，记录的列 c 值都是 '刘备'，这就是可重复读的含义。如果我们之后再把事务id为 200 的记录提交了，之后再回到刚才使用 REPEATABLE READ 隔离级别的事务中继续查找这个id为 1 的记录，得到的结果还是 '刘备'，具体执行过程大家可以自己分析一下。

5、MVCC下的读操作

1.当前读

在MVCC并发控制中，读操作可以分成两类：**快照读 (snapshot read)**与**当前读 (current read)**。

快照读：读取的是记录的可见版本 (有可能是历史版本)，不用加锁。

当前读：读取的是记录的最新版本，并且当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

在一个支持MVCC并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？

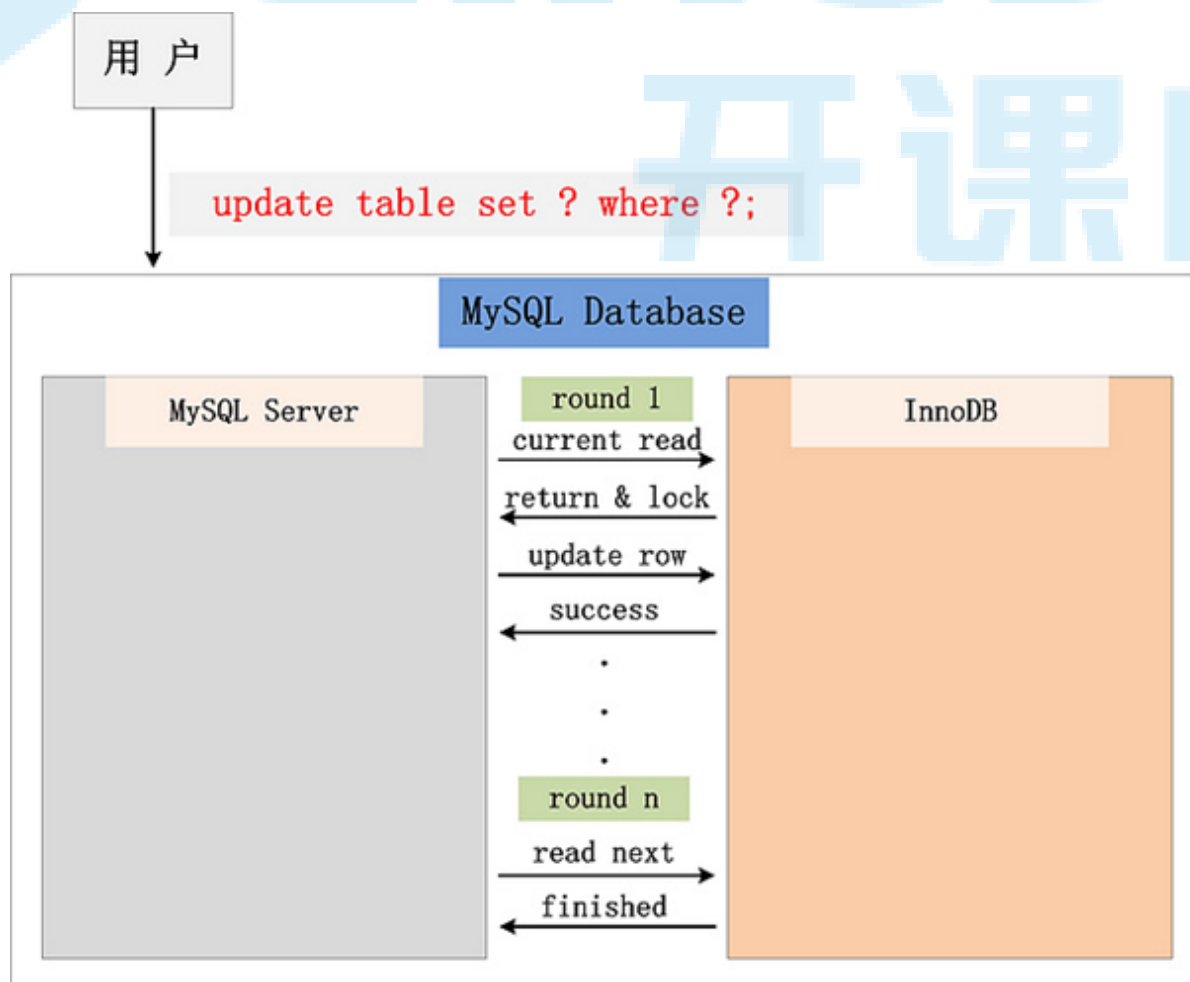
以MySQL InnoDB为例：

快照读：简单的select操作，属于快照读，不加锁。(当然，也有例外，下面会分析) 不加读锁 读历史版本

当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。加行写锁 读当前版本

```
1 select * from table where ? lock in share mode;
2 select * from table where ? for update;
3 insert into table values (...);
4 update table set ? where ?;
5 delete from table where ?;
```

所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加**S锁 (共享锁)**外，其他的操作，都加的是**X锁 (排它锁)**。



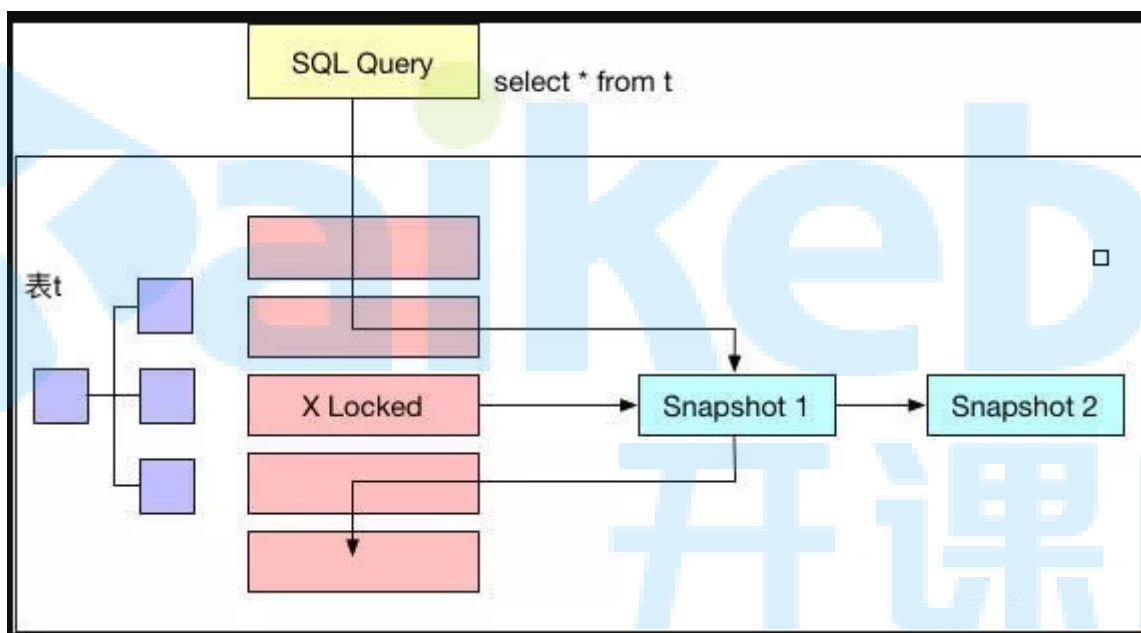
从图中，可以看到，一个Update操作的具体流程。当Update SQL被发给MySQL后，MySQL Server会根据where条件，读取第一条满足条件的记录，然后InnoDB引擎会将第一条记录返回，并加锁 (current read)。待MySQL Server收到这条加锁的记录之后，会再发起一个Update请求，更新这条记录。一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。因此，Update操作内部，就包含了一个当前读。同理，Delete操作也一样。Insert操作会稍微有些不同，简单来说，就是Insert操作可能会触发Unique Key的冲突检查，也会进行一个当前读。

注：根据上图的交互，针对一条当前读的SQL语句，InnoDB与MySQL Server的交互，是一条一条进行的，因此，加锁也是一条一条进行的。先对一条满足条件的记录加锁，返回给MySQL Server，做一些DML操作；然后在读取下一条加锁，直至读取完毕。

2.快照读

快照读也就是一致性非锁定读(consistent nonlocking read)是指InnoDB存储引擎通过多版本控制(MVCC)读取当前数据库中行数据的方式。

如果读取的行正在执行DELETE或UPDATE操作，这时读取操作不会因此去等待行上锁的释放。相反地，InnoDB会去读取行的一个最新可见快照。



会话A和会话B示意图：

会话A	会话B
BEGIN	
SELECT * FROM test WHERE id = 1;	
	BEGIN
	UPDATE test SET id = 3 WHERE id = 1;
SELECT * FROM test WHERE id = 1;	
	COMMIT;
SELECT * FROM test WHERE id = 1;	
COMMIT	

如上图所示，当会话B提交事务后，会话A再次运行 `SELECT * FROM test WHERE id = 1` 的SQL语句时，不同的事务隔离级别下得到的结果就不一样了。

6、MVCC总结

从上边的描述中我们可以看出来，所谓的MVCC（Multi-Version Concurrency Control，多版本并发控制）指的就是在使用 `READ COMMITTD`、`REPEATABLE READ` 这两种隔离级别的事务在执行普通的 `SELECT` 操作时访问记录的版本链的过程，这样子可以使不同事务的读-写、写-读操作并发执行，从而提升系统性能。

`READ COMMITTD`、`REPEATABLE READ` 这两个隔离级别的一个很大不同就是生成 `ReadView` 的时机不同，`READ COMMITTD` 在每一次进行普通 `SELECT` 操作前都会生成一个 `ReadView`，而 `REPEATABLE READ` 只在第一次进行普通 `SELECT` 操作前生成一个 `ReadView`，之后的查询操作都重复这个 `ReadView` 就好了。

四、事务回滚和数据恢复

[事务的隔离性由多版本控制机制和锁实现，而原子性、持久性和一致性主要是通过redo log、undo log和Force Log at Commit机制来完成的。](#) redo log用于在崩溃时恢复数据，undo log用于对事务的影响进行撤销，也可以用于多版本控制。而Force Log at Commit机制保证事务提交后redo log日志都已经持久化。

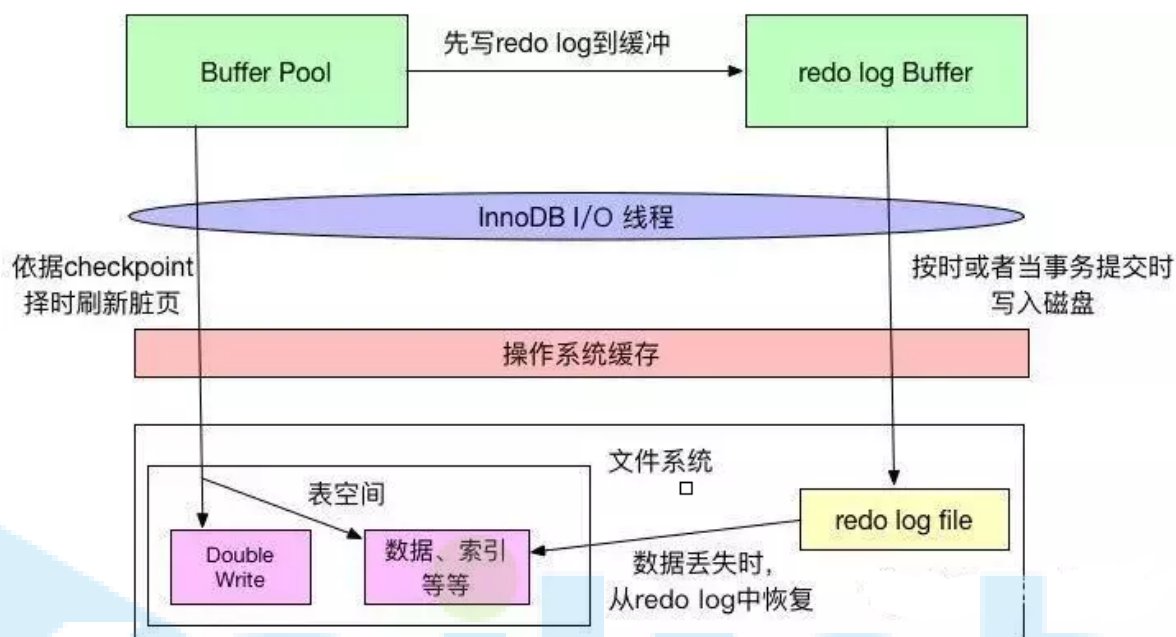
开启一个事务后，用户可以使用COMMIT来提交，也可以用ROLLBACK来回滚。其中COMMIT或者ROLLBACK执行成功之后，数据一定是会被全部保存或者全部回滚到最初状态的，这也体现了事务的原子性。但是也会有很多的异常情况，比如说事务执行中途连接断开，或者是执行COMMIT或者ROLLBACK时发生错误，Server Crash等，此时数据库会自动进行回滚或者重启之后进行恢复。

我们先来看一下redo log的原理，redo log顾名思义，就是重做日志，每次数据库的SQL操作导致的数据变化它都会记录一下，[具体来说，redo log是物理日志，记录的是数据库页的物理修改操作。](#)如果数据发生了丢失，数据库可以根据redo log进行数据恢复。

InnoDB通过Force Log at Commit机制实现事务的持久性，即当事务COMMIT时，必须先将该事务的所有日志都写入到redo log文件进行持久化之后，COMMIT操作才算完成。

当事务的各种SQL操作执行时，即会在缓冲区中修改数据，也会将对应的redo log写入它所属的缓存。当事务执行COMMIT时，与该事务相关的redo log缓冲必须都全部刷新到磁盘中之后COMMIT才算执行成功。

数据库日志和数据落盘机制，如下图所示：



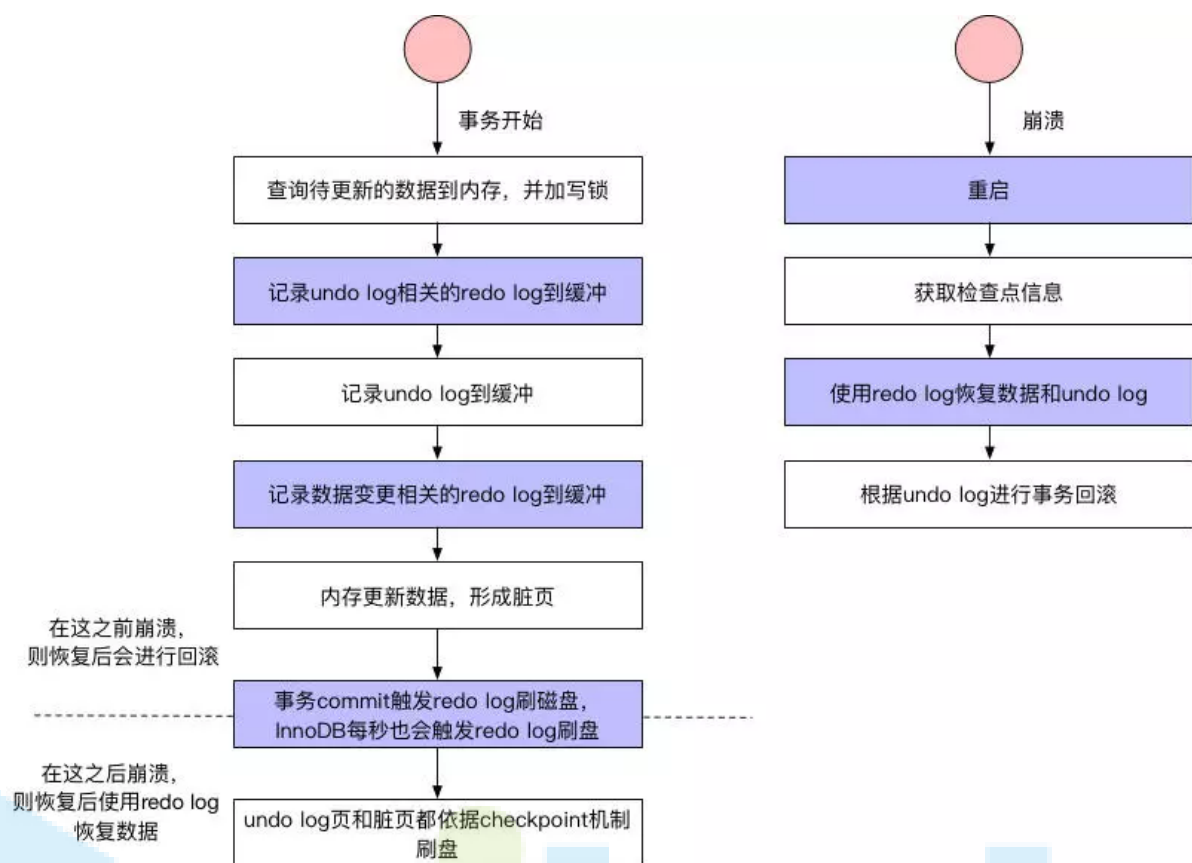
redo log写入磁盘时，必须进行一次操作系统的fsync操作，防止redo log只是写入了操作系统的磁盘缓存中。参数innodb_flush_log_at_trx_commit可以控制redo log日志刷新到磁盘的策略

数据库崩溃重启后需要从redo log中把未落盘的脏页数据恢复出来，重新写入磁盘，保证用户的数据不丢失。当然，在崩溃恢复中还需要回滚没有提交的事务。由于回滚操作需要undo日志的支持，undo日志的完整性和可靠性需要redo日志来保证，所以崩溃恢复先做redo恢复数据，然后做undo回滚。

在事务执行的过程中，除了记录redo log，还会记录一定量的undo log。undo log记录了数据在每个操作前的状态，如果事务执行过程中需要回滚，就可以根据undo log进行回滚操作。

undo log的存储不同于redo log，它存放在数据库内部的一个特殊的段(segment)中，这个段称为回滚段。回滚段位于共享表空间中。undo段中的以undo page为更小的组织单位。undo page和存储数据库数据和索引的页类似。因为redo log是物理日志，记录的是数据库页的物理修改操作。所以undo log（也看成数据库数据）的写入也会产生redo log，也就是undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护。如上图所示，表空间中有回滚段和叶节点段和非叶节点段，而三者都有对应的页结构。

我们再来总结一下数据库事务的整个流程，如下图所示。



事务的相关流程

事务进行过程中，每次DML sql语句执行，都会记录undo log和redo log，然后更新数据形成脏页，然后redo log按照时间或者空间等条件进行落盘，undo log和脏页按照checkpoint进行落盘，落盘后相应的redo log就可以删除了。此时，事务还未COMMIT，如果发生崩溃，则首先检查checkpoint记录，使用相应的redo log进行数据和undo log的恢复，然后查看undo log的状态发现事务尚未提交，然后就使用undo log进行事务回滚。事务执行COMMIT操作时，会将本事务相关的所有redo log都进行落盘，只有所有redo log落盘成功，才算COMMIT成功。然后内存中的数据脏页继续按照checkpoint进行落盘。如果此时发生了崩溃，则只使用redo log恢复数据。