

谈谈事务

C-2 创建:曹江勇, 最后修改: 曹江勇 11-15 17:43

目录

0.引入

1.1 事务起源

1.本地事务

1.1 原子性保证

1.1.1 事务的状态

1.1.2 如何保证原子性

1.2 事务隔离级别与一致性保证

1.2.1 解决并发事务带来的问题

1.2.2 事务隔离级别

1.2.3 mysql一致性保证

1.2.3.1 MVCC

1.2.3.1.1 版本链

1.2.3.1.2 ReadView

1.2.3.1.3 RC级别与RR级别下ReadView区别

1.2.4 锁

1.2.4.1 锁定读

1.2.4.2 多粒度锁

1.2.4.2.1 表锁

1.2.4.2.2 意向锁

1.2.4.2.2 行锁

2.分布式事务

2.1 分布式事务的使用场景

2.2 通用标准与分布式事务模型

2.2.1 DTP

2.2.2 XA规范

2.2.3 两阶段提交

2.3 分布式事务解决方案

2.3.1 TCC (Try-Confirm-Cancel)

2.3.2 MTXC/Seata

2.3.3 SAGA

参考

0.引入

本文尝试回答几个问题：

- 本地事务，这里主要涉及MySQL，事务机制是如何实现的，如何做ACID，平时说的MVCC、锁之类的到底是怎么实现的

- 分布式协同事务主要关注的是什么
- 随着SOA化、跨库、分库业务的越来越普遍，如何保证分布式业务的一致性

1.1 事务起源

事务(Transaction)是访问并可能更新数据库中各种数据项的一个程序执行单元(unit)。在关系数据库中，一个事务由一组SQL语句组成。事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

- 原子性 (atomicity)：一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性 (consistency)：事务必须是使数据库从一个一致性状态变到另一个一致性状态，事务的中间状态不能被观察到的。
- 隔离性 (isolation)：一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。隔离性又分为四个级别：读未提交(read uncommitted)、读已提交(read committed，解决脏读)、可重复读(repeatable read，解决虚读)、串行化(serializable，解决幻读)。
- 持久性 (durability)：持久性也称永久性 (permanence)，指一个事务一旦提交，它对数据库中数据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。

任何事务机制在实现时，都应该考虑事务的ACID特性，包括：本地事务、分布式事务

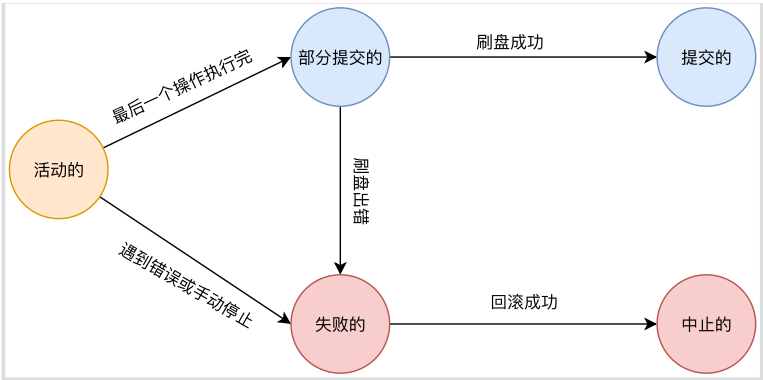
1.本地事务

1.1 原子性保证

1.1.1 事务的状态

数据库大概存在以下几个事务的状态：

- 活动的 (active)
事务对应的数据库操作正在执行过程中时，则称该事务处在活动的状态
- 部分提交 (partially committed)
当事务中的最后一个操作执行完成时，但由于操作都在内存中执行，所造成的影响并没有刷新到磁盘，我们将该事务状态称为部分提交
- 失败的 (failed)
当事务处于活动的或部分提交的状态时，如果遇到某些错误（数据库自身的错误、操作系统错误或直接断电等）而无法继续执行，或者人为停止当前事务的执行，我们就说该事务处于失败的状态
- 中止的 (aborted)
如果事务执行了半截而变为失败的状态，那么这时候应该已经对数据库中的数据造成了一些影响，而这些影响当然是要撤销的（也被称为回滚）。当回滚操作执行完毕后，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了中止状态
- 提交的 (committed)
当一个处于部分提交的事务将修改过的数据同步到磁盘上后，我们就说该事务处于提交的状态



原子性需要保证一个事务中的操作，要么全部执行成功，要么都不成功，那么如果在事务执行失败时，或者人为想撤回修改时，把已经做的进行回滚

1.1.2 如何保证原子性

(1) undo日志引入

为了保证事务执行失败或人为撤销修改时，对数据库的修改也撤回，则需要知道事务执行过程中，到底对数据库做了那些修改，也就是需要我们有机制保证每当我们对一条记录做改动时（这里的改动可以指INSERT、DELETE、UPDATE）把回滚时所需的東西都给记录下来，比如：

- 你插入一条记录时，至少要把这条记录的主键值记录下来，之后回滚的时候只需要把这个主键值对应的记录删掉就好了
- 你删除了一条记录，至少要把这条记录中的内容都记录下来，这样之后回滚时再把由这些内容组成的记录插入到表中就好了
- 你修改了一条记录，至少要把修改这条记录前的旧值都记录下来，这样之后回滚时再把这条记录更新为旧值就好了

而这些信息，就称为undo日志。

(2) 事务ID

由于一个事务会包含多个操作，则需要有一个东西串起事务执行的流程，这就是事务ID，如果某个事务执行过程中对某个表执行了增、删、改操作，那么InnoDB存储引擎会给它分配一个独一无二的事务id

好了，有了事务ID，那到底如何串起对记录的修改呢，下面看下事务ID到底会和记录有啥关系

这是一条行记录	记录的额外信息	row_id (非必需)	trx_id	roll_pointer	列数据
---------	---------	--------------	--------	--------------	-----

说明：

- 记录的额外信息：为了描述一条行记录而不得不加的额外信息，主要分为变长字段长度列表、NULL值列表和记录头信息三大类，与理解本文无关，可忽略
- row_id：隐藏列，如果行记录无主键或唯一索引，则InnoDB默认加的一个行唯一标识，与理解本文无关，可忽略
- trx_id：隐藏列，事务ID
- roll_pointer：指向undo日志的指针
- 列数据：真正的用户数据，与理解本文无关，可忽略

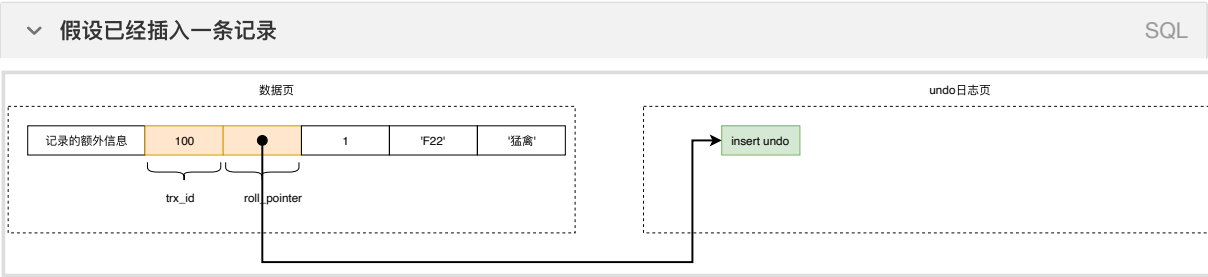
(3) undo日志与行记录的关系

实际上在设计InnoDB的事务机制时，分情况，有很多中undo日志，我们以update操作为例（这个Update不更新主键也不更新用户列占用空间的大小）

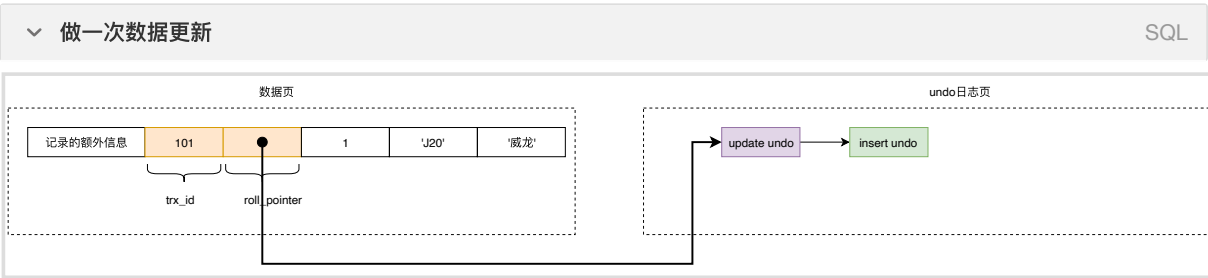
准备工作：

测试表

SQL



开始更新：



可以看出，执行完更新操作后，它对应的UNDO日志与插入的undo日志串成了一个链表，这初步形成了版本链

(4) 从undo日志格式来看版本链

更新数据undo日志中存储的信息

额外信息
old_roll_pointer (上一条undo日志指针)
主键列值
...
被更新列值
...

说明：

- 每条undo日志中均存在一个old_roll_pointer，指向当前undo日志的前一条日志（插入类型的undo日志的old_roll_pointer为空）
- undo日志会记录具体是那条记录被更新（主键列及值）
- undo日志会记载被更新列及对应的值

小结： 如果一个事务执行过程中出现错误，或者认为中止事务，mysql会基于当前事务ID，找到上一个版本的undo日志，执行事务回滚，保证原子性

Tips：通过undo日志串起的版本链不仅仅用于事务回滚，后面还有大用！

多唠叨几句关于undo日志，不看也行

1.2 事务隔离级别与一致性保证

1.2.1 解决并发事务带来的问题

并发访问相同记录往往容易带来数据不一致的问题，分情况来描述：

- 脏读（Dirty Read）：如果一个事务读到了另一个未提交事务修改过的数据，那就意味着发生了脏读
- 不可重复读（No-Repeatable Read）：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修改并提交后，该事务都能查询到最新值
- 幻读（Phantom）：如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先记录再次按照该条件查询时，能把另一个事务插入的记录也读出来，幻读强调的是在一个事务按照某个相同条件多次读取记录时，后读取时读到了之前没有读到的记录（其实这称之为不可重复读）
- 丢失更新：
 - 脏写：如果一个事务修改了另一个未提交事务修改过的数据
 - 第一类丢失更新：A事务撤销时，把已经提交的B事务的更新的事务覆盖了，这种错误可能会造成严重问题
 - 第二类丢失更新：A事务覆盖B事务已经提交的数据，造成B事务所做操作丢失

1.2.2 事务隔离级别

从前述并发执行事务带来的问题，按照严重程度：

脏写 > 脏读 > 不可重复读 > 幻读

我们知道MySQL是一个客户端 / 服务器架构的软件，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）。每个客户端都可以在自己的会话中向服务器发出请求语句，一个请求语句可能是某个事务的一部分，也就是对于服务器来说可能同时处理多个事务，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，我们既想保持事务的隔离性，又想让服务器在处理访问同一数据的多个事务时性能尽量高些，鱼和熊掌不可得兼，舍一部分隔离性而取性能者，SQL标准指定了四个隔离级别

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED（未提交读）	Possible	Possible	Possible
READ COMMITTED（已提交读）	Not Possible	Possible	Possible
REPEATABLE READ（可重复读）	Not Possible	Not Possible	Possible
SERIALIZABLE（串行化）	Not Possible	Not Possible	Not Possible

关于脏写，脏写这个问题太严重了，不论是哪种隔离级别，都不允许脏写的情况发生

1.2.3 mysql一致性保证

总结以上的并发事务带来的问题，主要有以下几种场景：

- 读-读：并发读相同的记录，这对记录没有什么影响
- 写-写：相同事务相继对相同记录做出改动，这种情况容易产生脏写

解决办法：加锁

- 读-写或写-读：一个事务进行读操作，另一个事务做写操作，这种并发访问容易产生脏读、不可重复读、幻读情况

解决办法：

- 读操作利用多版本并发控制（MVCC），写操作加锁

所谓MVCC就是通过在查询是生成ReadView，通过ReadView找到符合条件的记录版本（通过undo日志构建），其实类似于在生成ReadView的时候做了一次快照，查询语句只能读到在ReadView之前已提交的修改，在生成ReadView之前未提交的事务或者之后才开启的事务所做的更改是看不到的。而写操作肯定针对的是最新版本的记录，读记录的历史版本和改动记录的最新版本不冲突，也就是采用MVCC时，读-写操作并不冲突

- 读、写操作都采用加锁的方式

很明显，采用MVCC的话，读-写操作彼此不冲突，性能更高，采用加锁方式的话，读-写操作彼此需要排队执行，影响性能。一般情况下我们当然愿意采用MVCC来解决读-写操作并发执行的问题

多说一句：这里的读，我们认为是普通的读数据方式（后面会提到的一致性读），mysql还提供特殊的读数据方式，读数据存在一定的独占性（这种方式后面会提到，称为锁定读），锁定读实际上可以配合写操作来解决丢失更新的问题

1.2.3.1 MVCC

1.2.3.1.1 版本链


前面介绍了对InnoDB存储引擎来说，行记录存在两个必要的隐藏列：

- trx_id：每次一个事务对某条聚簇索引记录进行改动时，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，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

下面以这个表为例介绍：

dynasty_demoSQL

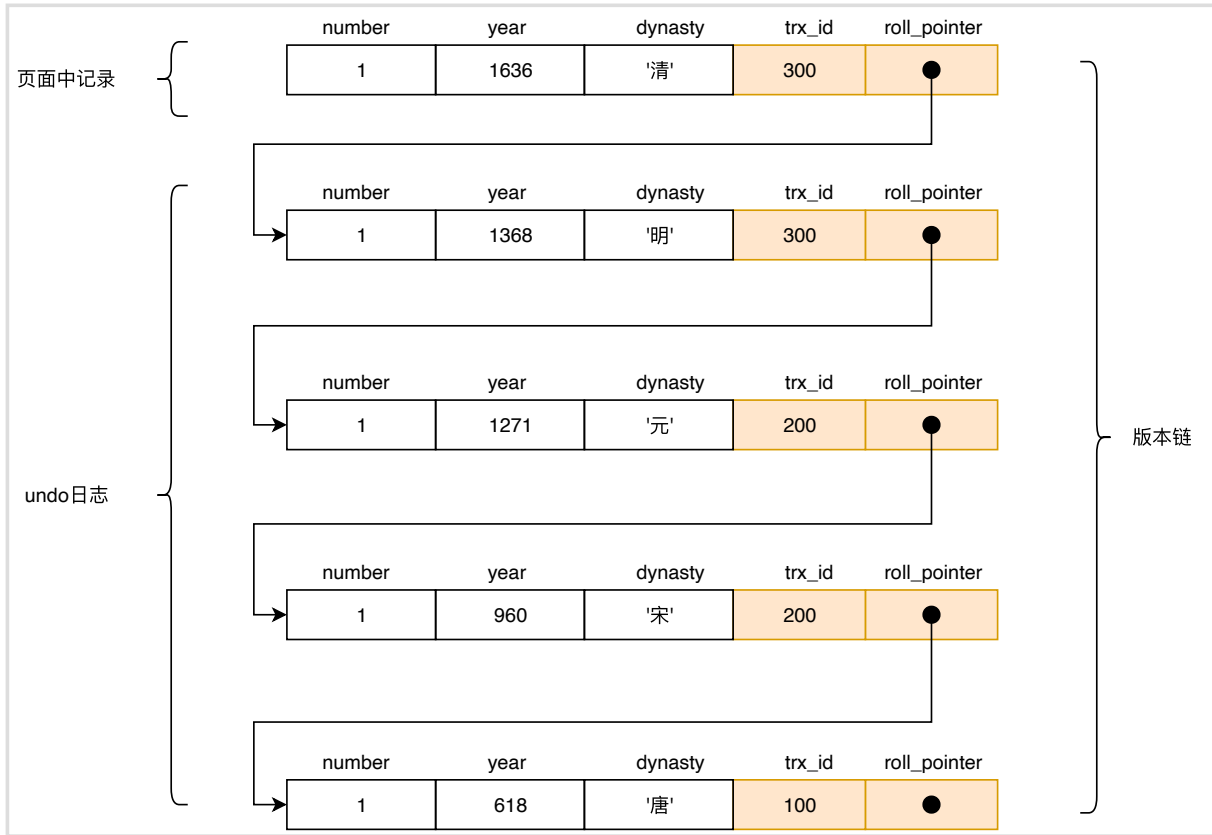
插入数据SQL

number	year	dynasty	trx_id	roll_pointer
1	618	'唐'	80	

假设后面事务200和300对这条记录做了update

时间编号	trx 200	trx
1	BEGIN;	
2		BEC
3	UPDATE dynasty_demo SET year=960, dynasty = '宋' where number = 1;	
4	UPDATE dynasty_demo SET year=1271, dynasty = '元' where number = 1;	
5	commit;	

6		UPI
7		UPI
8		con



对该记录的每次更新后，都会将旧值放到一条undo日志中，就算是该记录的一个版本，随着更新次数的增多，所有的版本都会被roll_pointer属性连接成一个链表，我们把这个链表称之为版本链，版本链的头节点就是当前记录最新的值。另外，每个版本中还包含生成该版本时对应的事务id

1.2.3.1.2 ReadView

对于使用READ UNCOMMITTED隔离级别的事务来说，由于可以读取到未提交的事务修改过的记录，所以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，设计innoDB的大叔规定使用加锁的方式来访问记录；对于READ COMMITTED和REPEATABLE READ隔离级别的事务来说，都必须保证读到已经提交了的事务修改的记录，也就是说假如另一个事务已经修改了记录但是尚未提交，是不能直接读取最新版本的记录的，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，为此引入ReadView这个概念：

- m_ids：表示生成ReadView时当前系统中活跃的读写事务的事务id列表
 - min_trx_id：表示在生成ReadView时当前系统中活跃的读写事务中最小的事务ID，也就是min_ids的最小值
 - max_trx_id：表示生成ReadView时系统中应该分配给下一个事务的id值
- max_trx_id并不是m_ids里的最大值，事务ID是递增分配的，比如现在有1，2，3三个事务，之后是事务ID为3的事务提交了。那么一个新的读事务在生成ReadView时，m_ids就包括1和2，min_trx_id的值就是1，max_trx_id的值就是4
- creator_trx_id：表示生成该ReadView的事务的事务ID

只有在对表中的记录做改动时（执行INSERT、DELETE、UPDATE这些语句时）才会为事务分配事务id，否则在一个只读事务中的事务id值都默认为0。

有了ReadView，可以按照以下步骤进行版本控制：

- 如果被访问版本的trx_id属性与ReadView中的creator_trx_id相同，意味着当前事务在访问自己修改的记录，所以该版本可以被当前事务访问
- 如果被访问版本的trx_id属性小于ReadView中的min_trx_id值，表明生成该版本的事务在当前事务生成ReadView前已经提交了，所以该版本可以被当前事务访问
- 如果被访问版本的trx_id属性大于等于ReadView中的max_trx_id值，表明生成该版本的事务在当前事务生成ReadView后才开启，所以该版本不可被当前事务访问
- 如果被访问版本的trx_id属性在ReadView的min_trx_id和max_trx_id之间，那就需要判断trx_id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是活跃的，该版本不可被访问；如果不在，说明创建ReadView时生成该版本的事务已被提交，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，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完全不可见，查询结果就不包含该记录

1.2.3.1.3 RC级别与RR级别下ReadView区别

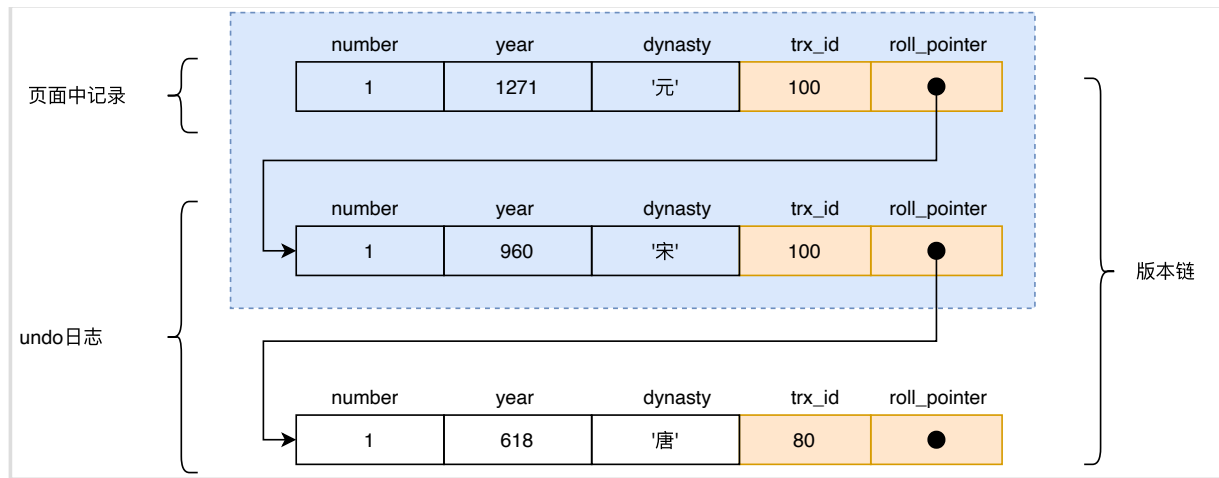
前面说过，RR级别可以在一定程度控制幻读，这个实际上是通过MVCC来实现的，那同样都是MVCC实现，两个事物隔离级别是如何达到不一样的数据一致性要求呢？答案实际也很简单，就是生成ReadView的时机

(1) READ COMMITTED——每次读取数据都生成一个ReadView

假设现在系统中有两个事务ID分为100和200的事务执行

```
^ 代码块 SQL
1  # Transaction 100
2  BEGIN;
3
4  UPDATE dynasty_demo SET year=960, dynasty = '宋' where number = 1;
5
6  UPDATE dynasty_demo SET year=1271, dynasty = '元' where number = 1;
7
8  # Transaction 200
9  BEGIN;
10
11 # 更新了一些别的表的记录
12 ...
```

此时dynasty_demo中的number为1的版本链为：



假设有一个RC隔离级别的查询：

代码块

SQL

```

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 dynasty_demo WHERE number = 1; 得到的结果是year=618 dynasty='唐'

```

说明：

这个SELECT1的执行过程如下：

- 在执行SELECT语句时会生成一个ReadView，ReadView的m_ids列表就是[100,200]，min_trx_id就是100，max_trx_id为201，creator_trx_id为0；
- 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dynasty是'元'，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dynasty的内容是'宋'，该版本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dynasty的内容是'唐'，该版本的trx_id值为80，小于ReadView中的min_trx_id值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dynasty为'唐'的记录

之后，我们把事务ID为100的事务和200的事务提交一下

trx 100提交

SQL

```

1  # Transaction 100
2  BEGIN;
3
4  UPDATE dynasty_demo SET year=960, dynasty = '宋' where number = 1;
5
6  UPDATE dynasty_demo SET year=1271, dynasty = '元' where number = 1;
7
8  commit;

```

然后我们在trx200也做下修改，但是不提交

trx 200提交

SQL

```

1  # Transaction 200
2  BEGIN;

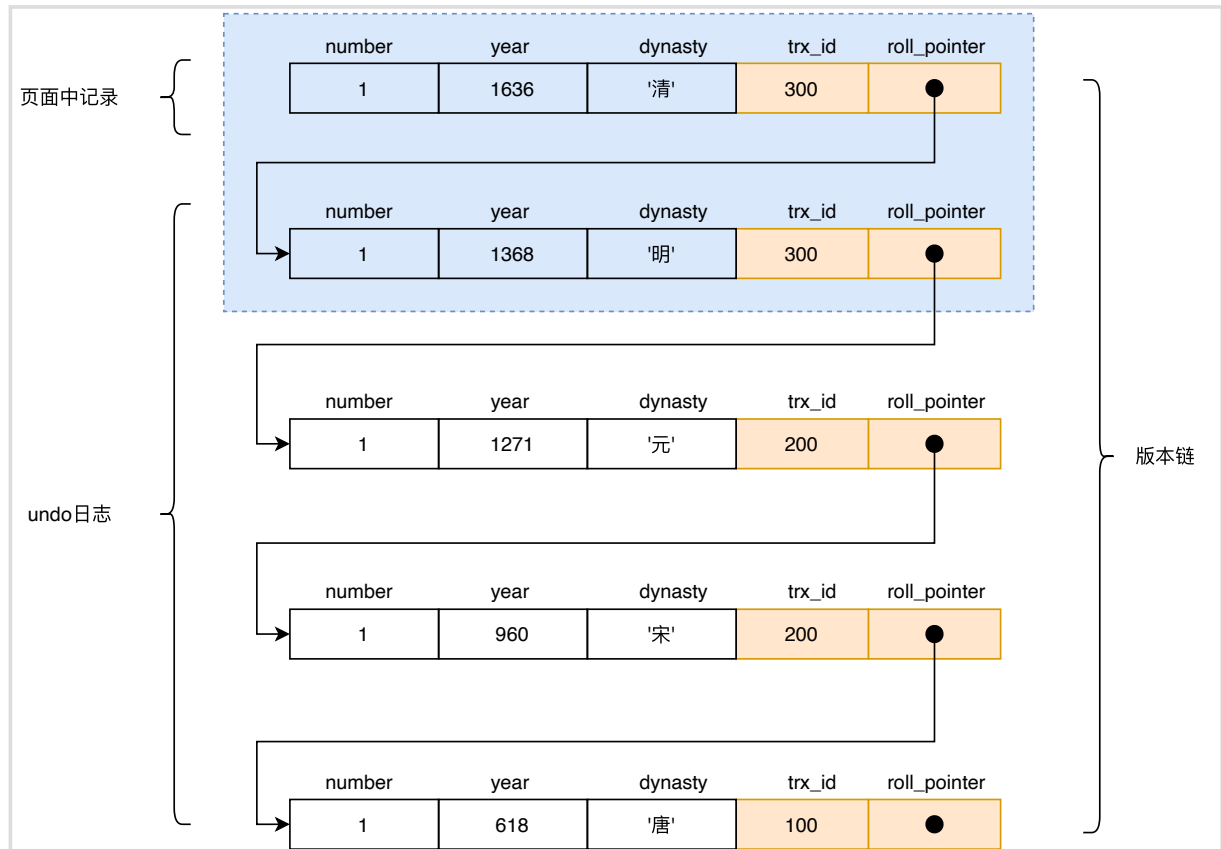
```

```

3
4 # 更新了一些别的表的记录
5 ...
6
7 UPDATE dynasty_demo SET year=1368, dynasty = '明' where number = 1;
8
9 UPDATE dynasty_demo SET year=1636, dynasty = '清' where number = 1;

```

此时再来看下number=1的记录



然后再在RC级别下，查下number=1的记录

代码块

SQL

```

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 dynasty_demo WHERE number = 1; 得到的结果是year=618 dynasty='
6
7 # SELECT2: Transaction 100提交, Transaction 200未提交
8 SELECT * FROM dynasty_demo WHERE number = 1; # 得到的列dynasty的值为'元'

```

这次，我们执行的是SELECT2，分析下：

- 在执行SELECT语句时会又单独生成一个ReadView，该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，所以再次生成快照时就没有它了），`min_trx_id`为200，`max_trx_id`为201，`creator_trx_id`为0。
- 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版的`dynasty`值是'清'，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dynasty的内容是'明'，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dynasty的内容是'元'，该版本的trx_id值为100，小于ReadView中的min_trx_id值200，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dynasty为'元'的记录

以此类推，如果之后事务id为200的记录也提交了，再次在使用READ COMMITTED隔离级别的事务中查询表hero中number值为1的记录时，得到的结果就是'清'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 —— 在第一次读取数据时生成ReadView

📌 RR级别：对于使用**REPEATABLE READ**隔离级别来说，只会在第一次查询语句时生成ReadView

不再详细分析，按照时间线分析如下

时间编号	trx 100	trx 200
1	BEGIN;	
2		BEGIN;
3	UPDATE dynasty_demo SET year=960, dynasty = '宋' where number = 1;	
4	UPDATE dynasty_demo SET year=1271, dynasty = '元' where number = 1;	
5		
6	commit;	
7		UPDATE dyn
8		UPDATE dyn
9		commit;
10		

📌 小结：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使用这个ReadView就好了。

1.2.4 锁

通过MVCC我们基本可以解决读-写/写-读的并发事务带来的数据一致性问题，那么接下来我们需要考虑的是写-写场景下，如何解决并发事务的数据隔离性和一致性，以及我们在某些场景下为了解决丢失更新的问题，这种都是通过加锁来解决，在介绍锁之前，按照资源是否共享，将锁分类：

- 共享锁（Shared Locks），简称S锁。在事务读取一条记录时，需要先获取该记录的S锁
- 独占锁（Exclusive Locks），简称X锁。在事务改动一条记录时，需要先获取该记录的X锁

兼容性	X	S
X	不兼容	不兼容
S	不兼容	兼容

Tips：这里实际上说的是行锁

1.2.4.1 锁定读

支持对读操作加锁的查询语句：

^ 加读共享锁 Java

```
1  SELECT ... LOCK IN SHARE MODE;
```

^ 加读共享锁 Java

```
1  SELECT ... FOR UPDATE;
```

抛转引玉，介绍个用法：

假定存在多个服务更新某个表的字段，更新的逻辑都是先读某个字段，然后在其上进行修改，为了避免第一类丢失更新

^ 代码块 Java

```
1  begin;
2  select * from MARS_Status where launchId=123456;
3  update MARS_Status set status=1 where planId=123456;
4  commit;
```

Tips：这实际上是悲观锁的实现方式，由于使用了排他锁，可能会对性能有影响；所以，在业务代码真正的实现里，我们可以考虑使用乐观锁来实现

1.2.4.2 多粒度锁

InnoDB既支持表锁，也支持行锁；表锁粒度较粗，但是实现简单，当然我们在真正的操作过程中，只需要锁几条记录，我们使用表锁没有必要

1.2.4.2.1 表锁

在对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，innoDB是不会为这个表添加表级的S锁或X锁。

另外，对某个表执行ALTER TABLE、DROP TABLE这类的DDL时，其他事务对这个表并发执行SELECT、INSERT、DELETE、UPDATE的语句会发生阻塞，同理，某个事务中对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，在其他会话中对这个表执行DDL语句也会发生阻塞。这个过程其实是通过在server层使用一种称之为元数据锁（英文名：Metadata Locks，简称MDL）来实现的。个人感觉，InnoDB存储引擎的表级S锁和X锁比较鸡肋（只在一些特殊场景会用到，比如崩溃恢复过程会用到）

1.2.4.2.2 意向锁

前面我们说了表锁，但是加表锁是有条件的，比如，想对表上S锁，则需要保证表中没有行X锁；比如像对表上X锁，则需要保证表中没有行X锁和行S锁；那么如何确定行中没有X锁和S锁呢，遍历是不可能遍历的；那么就提出了一个意向锁（Intention Locks）的概念：


- 意向共享锁（Intention Shared Lock），简称IS锁。当事务准备在某条记录上加S锁时，需要先在表级别加一个IS锁。
- 意向独占锁（Intention Exclusive Lock，简称IX锁）。当事务准备在某条记录上加X锁时，需要先在表级别加一个IX锁

所以回到刚才的问题上

- 如果想对表上S锁，则需要保证表中没有行X锁，怎么办呢，看表是否存在IX锁，因为如果表中存在行记录加了X锁的话，肯定会先加个表级IX锁
- 如果想对表上X锁，则需要保证表中没有行X锁和行S锁，怎么办呢？看表是否存在IS锁或IX锁，因为如果表中存在行记录加了S锁或X锁的话，肯定会先加表级IS锁和IX锁

看下表级别各个锁的兼容性

兼容性	X	IX	S	IS
X	不兼容	不兼容	不兼容	不兼容
IX	不兼容	兼容	不兼容	兼容
S	不兼容	不兼容	兼容	兼容
IS	不兼容	兼容	兼容	兼容

 总结一下：IS、IX锁是表级锁，它们的提出仅仅为了在之后加表级别的S锁和X锁时可以快速判断表中的记录是否被上锁，以避免用遍历的方式来查看表中有没有上锁的记录，也就是说其实IS锁和IX锁是兼容的，IX锁和IX锁是兼容的

1.2.4.2.2 行锁

我们平常更多见到的应该是行锁，也就是某条记录上的锁，下面介绍下比较常见的行锁：

(1) Record Locks

Record Locks就是记录锁，仅仅是把一条记录锁上，官方的名字是LOCK_REC_NOT_GAP，为了区别后面的行锁，我们简称它为基本锁（本来想叫普通锁，奈何后面的next-key锁官方名字有ORDINARY😂）

	为number=8的行记录加 LOCK_REC_NOT_GAP锁				
number列	1	3	8	15	20
year列	618	960	1271	1368	1636
dynasty列	'唐'	'宋'	'元'	'明'	'清'

基本锁是有S锁和X锁之分的，让我们分别称之为S型基本锁和X型基本锁，对同一条记录加锁时，满足前面对行锁的兼容性原则

(2) Gap锁

之前说MySQL的**REPEATABLE READ**隔离级别可以解决幻读问题，解决方案有两种，可以使用MVCC解决，也可以使用加锁来解决。但是事务在首次执行读取数据时，那条幻影记录尚不存在，无法加基本锁，所以提出了Gap Locks锁，官方的类型名为LOCK_GAP，简称gap锁。

	为number=8的行记录加 LOCK_GAP锁				
number列	1	3	8	15	20
year列	618	960	1271	1368	1636
dynasty列	'唐'	'宋'	'元'	'明'	'清'

以上的是在number范围(3,8)加了gap锁，表示在这个范围内不允许插入，当然不是一直不允许插入，如果gap锁所在事务提交后，number(3,8)的新纪录可以插入

注意：**gap**锁仅仅是为了防止插入幻影记录而提出的，如果你对一条记录加了**gap**锁（不论是共享**gap**锁还是独占**gap**锁），并不会限制其他事务对这条记录加基本锁或者继续加**gap**锁

所以gap锁管的是间隙，目的是为了防止幻读

(3) Next-Key Locks

有时候我们想既锁住某条记录，又想阻止其他事务在该记录前面的间隙插入新记录，设计了Next-Key Locks，官方名称为LOCK_ORDINARY，我们也可以简称为next-key锁，比方说我们把number值为8的那条记录加一个next-key锁的示意图如下：

为number=8的行记录加 LOCK_ORDINARY锁

number列	1	3	8	15	20
year列	618	960	1271	1368	1636
dynasty列	'唐'	'宋'	'元'	'明'	'清'

next-key锁的本质就是一个基本锁和一个gap锁的合体，它既能保护该条记录，又能阻止别的事务将新记录插入被保护记录前边的间隙

(4) 插入意向锁

一个事务在插入一条记录时需要判断一下插入位置是不是被别的事务加了所谓的gap锁（next-key锁也包含gap锁），如果有的话，插入操作需要等待，直到拥有gap锁的那个事务提交；InnoDB在设计时，提出这种锁等待也需要设置锁接口，表明有事务想在某个间隙中插入新记录，但是现在在等待。InnoDB的就把这种类型的锁命名为Insert Intention Locks，官方的类型名称为：LOCK_INSERT_INTENTION，我们简称为插入意向锁

为number=8的行记录加 LOCK_INSERT_INTENTION锁

number列	1	3	8	15	20
year列	618	960	1271	1368	1636
dynasty列	'唐'	'宋'	'元'	'明'	'清'

下面简单介绍下加锁相关的原则和分析

- 一些原则：

加锁分析的时候，我们需要设定一些前提：

- 事务的隔离级别：隔离级别不同，对事务的隔离性和一致性要求不同，加锁逻辑必然也不相同，比如gap锁是RR级别为了防止幻读而设置的，那么RC就不会有
- 语句执行时用的索引，执行时用的索引不同，可能会导致加锁顺序的不同
- 查询条件
- 具体执行的语句类型

- 一些建议：

- 进行更新操作时，要注意查询条件以及会更新的列，比如通过主键更新数据，一般来说只会对应的主键索引加锁，那是如果更新的数据列中包含索引列，那还会对对应的索引列加锁，而如果同时有其他事务已经对相关二级索引列加锁了，那可能会造成锁等待，严重的可能会产生死锁；所以建议加锁的时候，同一个业务逻辑，查询条件和更新列尽量相同

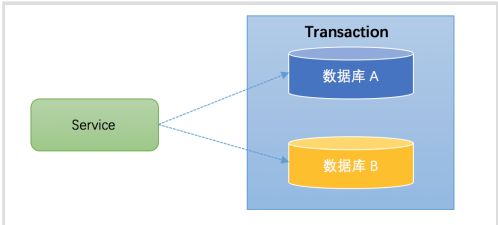
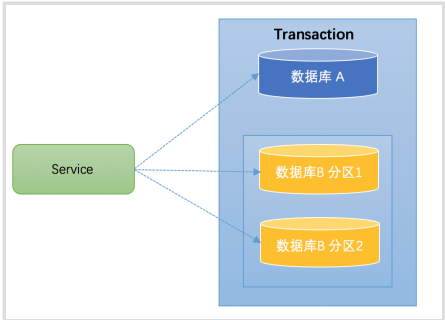

- 不建议加大范围的Gap锁（select * from xxx where id > xxx lock in share mode），这样很容易造成资源阻塞

2.分布式事务

分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。

本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

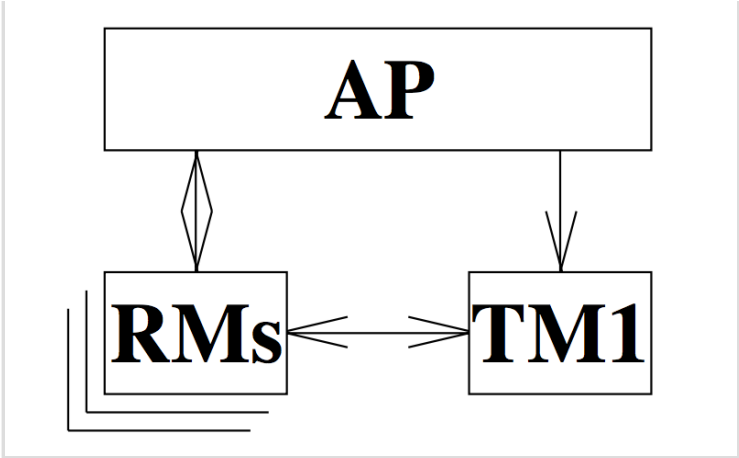
2.1 分布式事务的使用场景

	跨库事务	分库分表	服务
架构示例			
说明	跨库事务指的是，一个应用某个功能需要操作多个库，不同的库中存储不同的业务数据	通常一个库数据量比较大或者预期未来的数据量比较大，都会进行水平拆分，也就是分库分表	现行-服务-的通信
关键点	如果需要一个操作，涉及的多个库的数据插入，要么都成功，要么都失败，那么就面临分布式事务问题	对于sql: insert into user(id,name) values (1,"zhangsan"),(2,"lisi"), 现在进行了分库分表，如果希望将1号记录插入分库1，2号记录插入分库2。所以数据库中间件要将其改写为2条sql，分别插入两个不同的分库，此时要保证两个库要不都成功，要不都失败，因此基本上所有的数据库中间件都需要考虑着分布式事务的问题。	在执行功，且

2.2 通用标准与分布式事务模型

2.2.1 DTP

X/Open为分布式事务制定了一些标准，其提出了分布式事务处理(Distributed Transaction Processing,简称DTP)

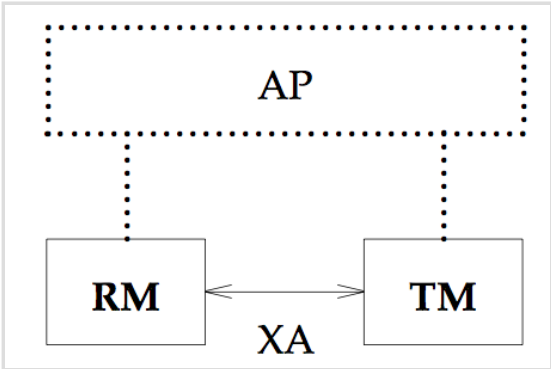


其中：

- 应用程序(Application Program ， 简称AP)：用于定义事务边界(即定义事务的开始和结束)，并且在事务边界内对资源进行操作。
- 资源管理器(Resource Manager， 简称RM)：如数据库、文件系统等，并提供访问资源的方式。
- 事务管理器(Transaction Manager ， 简称TM)：负责分配事务唯一标识，监控事务的执行进度，并负责事务的提交、回滚等。

2.2.2 XA规范

XA规范的最主要的作用是，就是定义了RM-TM的交互接口，下图更加清晰了演示了XA规范在DTP模型中发挥作用的位置

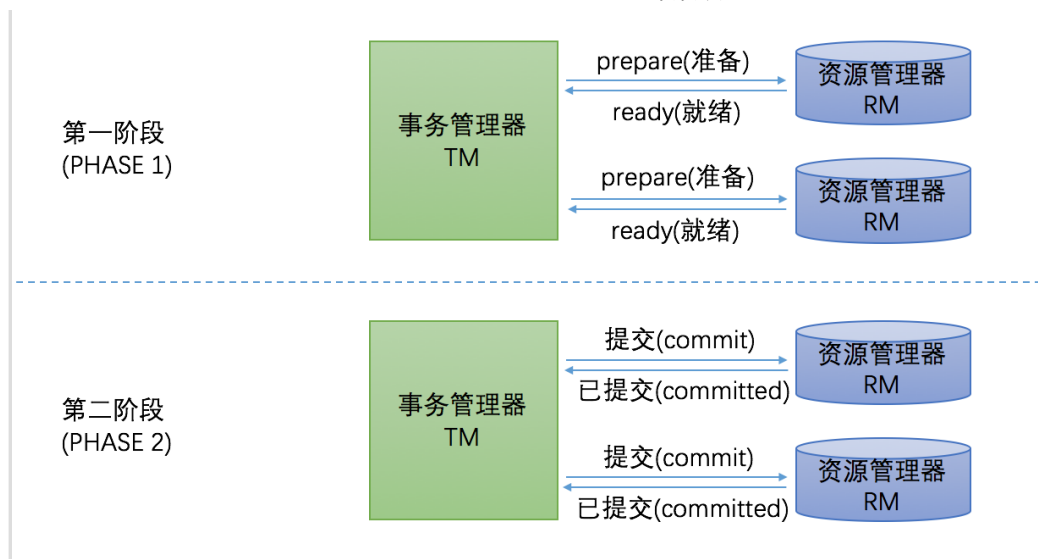


说明：

- 在DTP参考模型(<<Distributed Transaction Processing: Reference Model>>)中，指定了全局事务的提交要使用two-phase commit协议；
- 而XA规范(<< Distributed Transaction Processing: The XA Specification>>)只是定义了两阶段提交协议中需要使用到的接口，也就是上述提到的RM-TM交互的接口，因为两阶段提交过程中的参与方，只有TM和RMs。

2.2.3 两阶段提交

两阶段提交，顾名思义，将提交分为两个阶段，一阶段做提交准备；一阶段做提交



说明：

在阶段1：TM通知各个RM准备提交它们的事务分支。如果RM判断自己进行的工作可以被提交，那就就对工作内容进行持久化，再给TM肯定答复；要是发生了其他情况，那给TM的都是否定答复。在发送了否定答复并回滚了已经的工作后，RM就可以丢弃这个事务分支信息。

以mysql数据库为例，在第一阶段，事务管理器向所有涉及到的数据库服务器发出prepare"准备提交"请求，数据库收到请求后执行数据修改和日志记录等处理，处理完成后只是把事务的状态改成"可以提交"，然后把结果返回给事务管理器。

在阶段2：TM根据阶段1各个RM prepare的结果，决定是提交还是回滚事务。如果所有的RM都prepare成功，那么TM通知所有的RM进行提交；如果有RM prepare失败的话，则TM通知所有RM回滚自己的事务分支。

以mysql数据库为例，如果第一阶段中所有数据库都prepare成功，那么事务管理器向数据库服务器发出"确认提交"请求，数据库服务器把事务的"可以提交"状态改为"提交完成"状态，然后返回应答。如果在第一阶段内有任何一个数据库的操作发生了错误，或者事务管理器收不到某个数据库的回应，则认为事务失败，回滚所有数据库的事务。数据库服务器收不到第二阶段的确认提交请求，也会把"可以提交"的事务回滚。

❗ 基于XA规范实现存在问题：

- 在准备阶段，持有事务状态，对资源锁定；一直到第二阶段才提交；对于参与系统多的场景；会导致参与系统阻塞，这种长事务场景很影响性能
- RM需要实现XA接口，很多场景下RM就是DB，但是目前主流数据库对XA的支持都不够好
- 事务管理器与RM交互，存在单点风险
- ...

✓ 唠叨几句

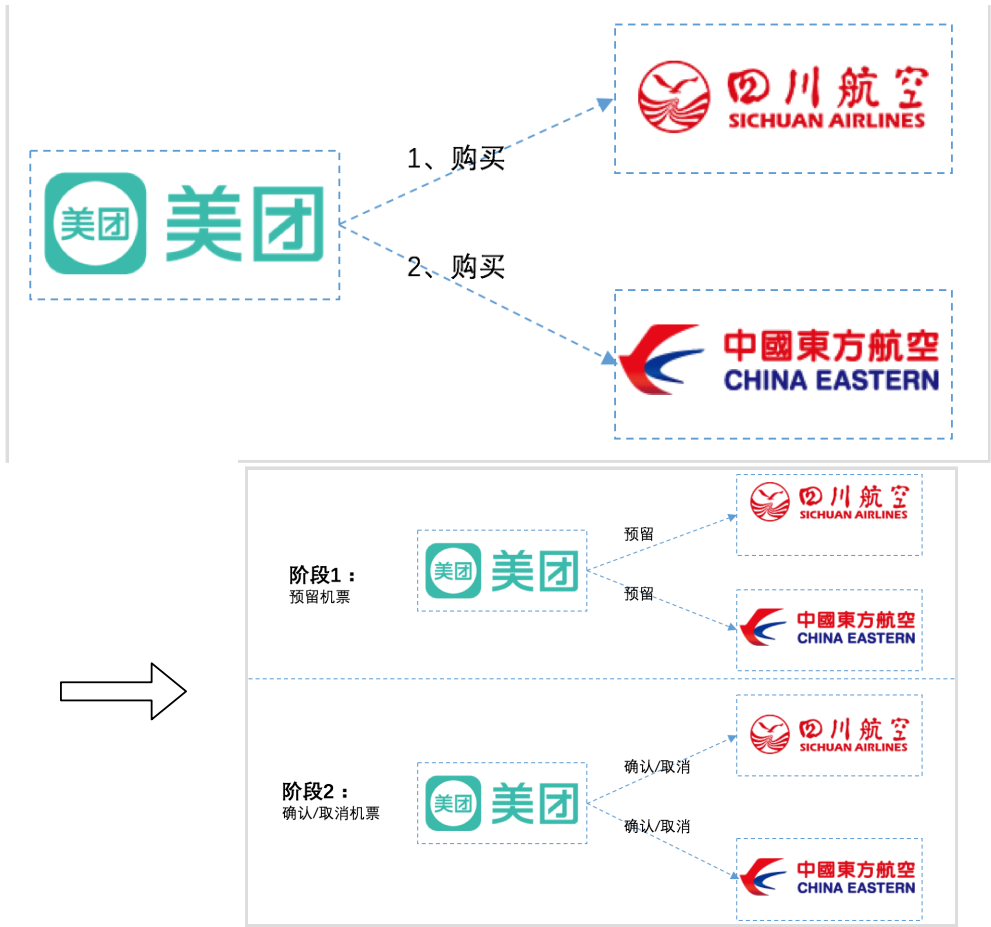
2.3 分布式事务解决方案

2.3.1 TCC (Try-Confirm-Cancel)

(1) 简介

TCC是Try-Confirm-Cancel的简称，作用主要是解决跨服务调用场景下的分布式事务问题

简单的例子



说明：

已美团购联程票为例，想分别预定川航和东航的机票，但是行程必然是需要连续都成功的（一个成功，一个不成功，怎么飞🤔），所以需要保证这两个流程要么都成功、要么都不成功，按照TCC的方式，可以拆解为：

Try阶段：

完成所有业务检查（一致性），预留业务资源(准隔离性)

上面航班预定案例的阶段1，机票就是业务资源，所有的资源提供者(航空公司)预留都成功，try阶段才算成功

Confirm阶段：

确认执行业务操作，不做任何业务检查，只使用Try阶段预留的业务资源。上面航班预定案例的阶段2，美团APP确认两个航空公司机票都预留成功，因此向两个航空公司分别发送确认购买的请求。

Cancel阶段：

取消Try阶段预留的业务资源。回顾上面航班预定案例的阶段2，如果某个业务方的业务资源没有预留成功，则取消所有业务资源预留请求。

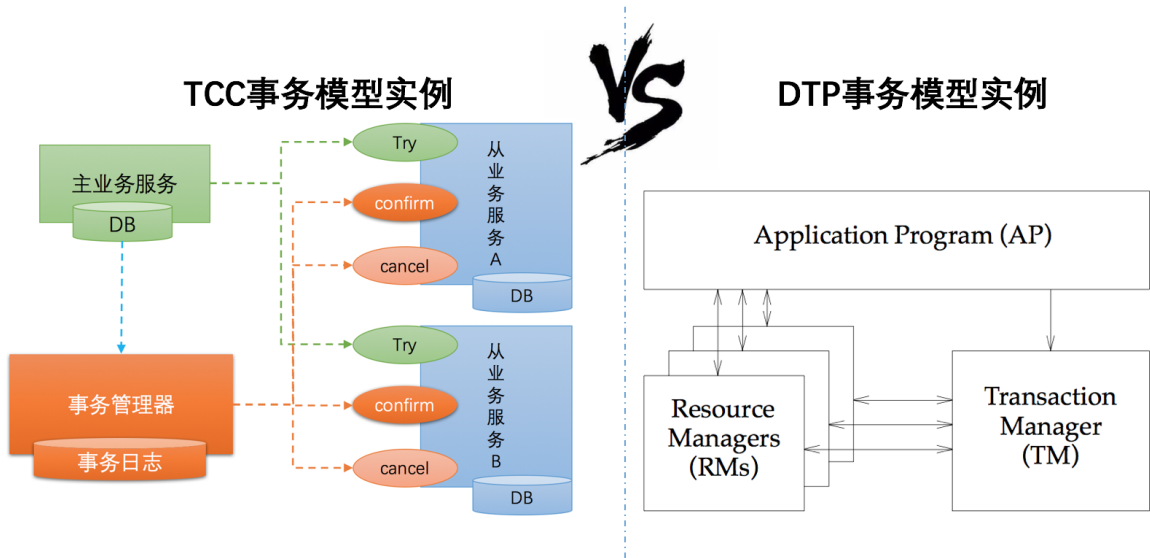
(2) TCC与XA的简单对比

- 相同点：都是两阶段提交：TCC是第一阶段进行资源预留，第二阶段根据资源预留结果进行确认或取消；XA第一阶段是各分支事务进行事务提交准备，第二阶段根据第一阶段的结果，进行事务提交或回滚
- 不同点：
 - XA是强一致性的，XA在第一阶段到第二阶段提交或回滚之前，是需要一直持有资源的，这个在可用性上和性能上是个硬伤
 - TCC是最终一致性，由于是每个参与者各自执行try和commit方法，各自处理各自的业务，不会涉及一直的资源加锁；

✓ 是不是还没看出两者的区别😂，我尝试再解释下

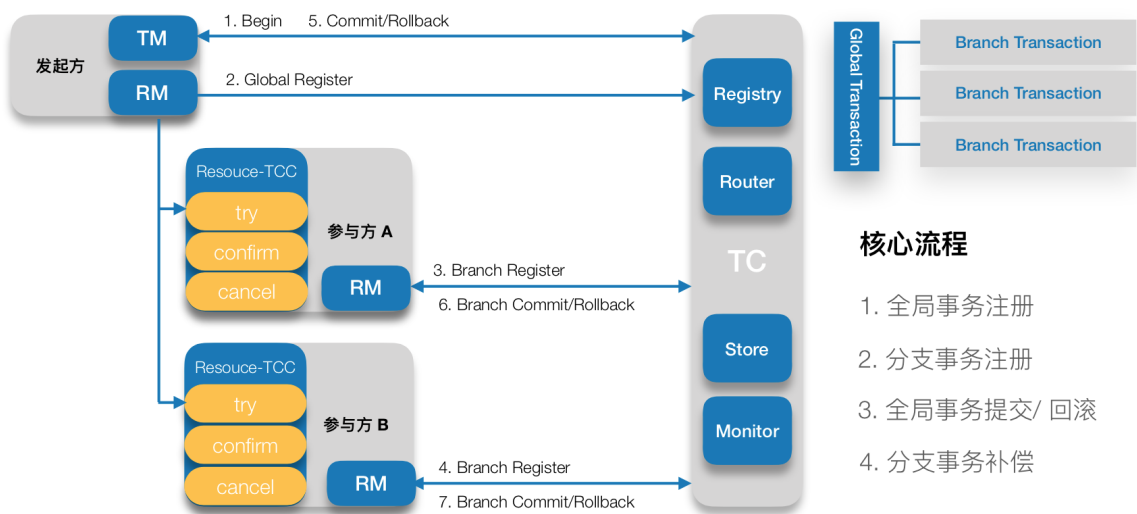
(2) 实现

- 事务模型：



- 实现

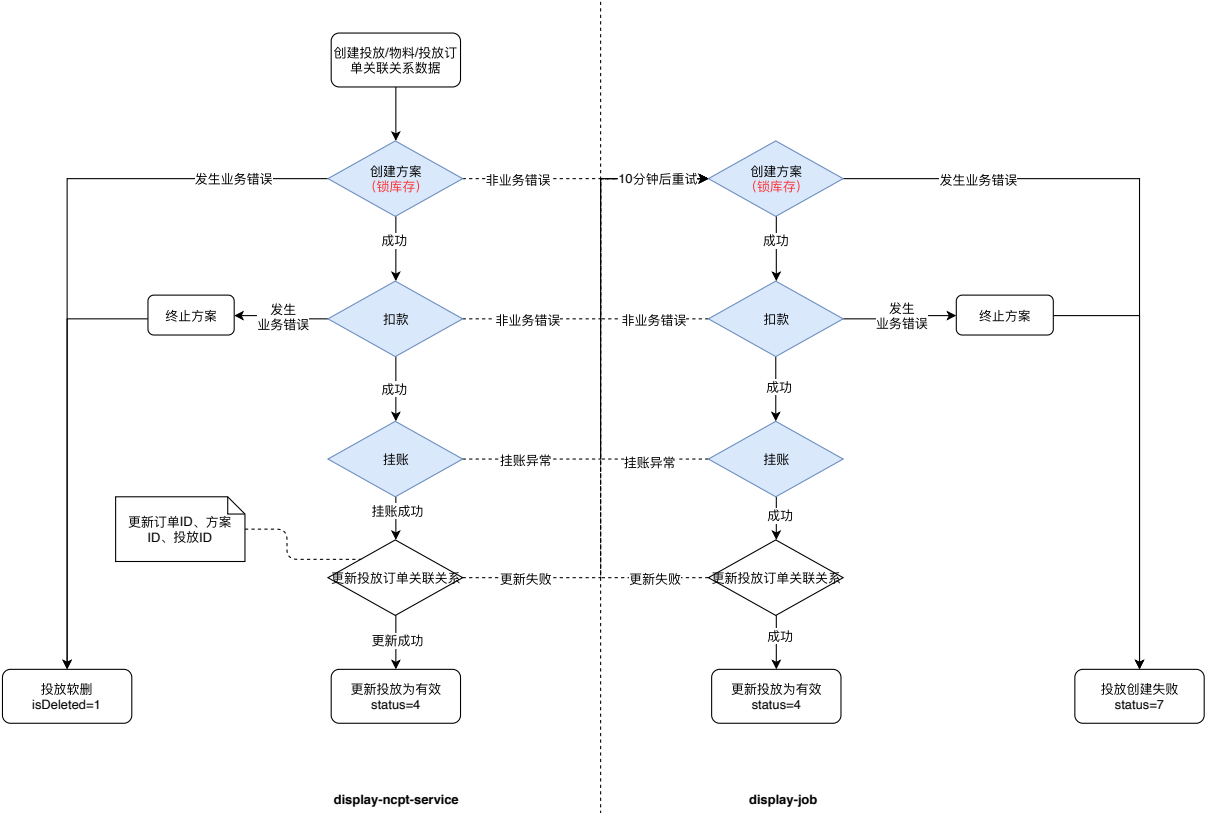
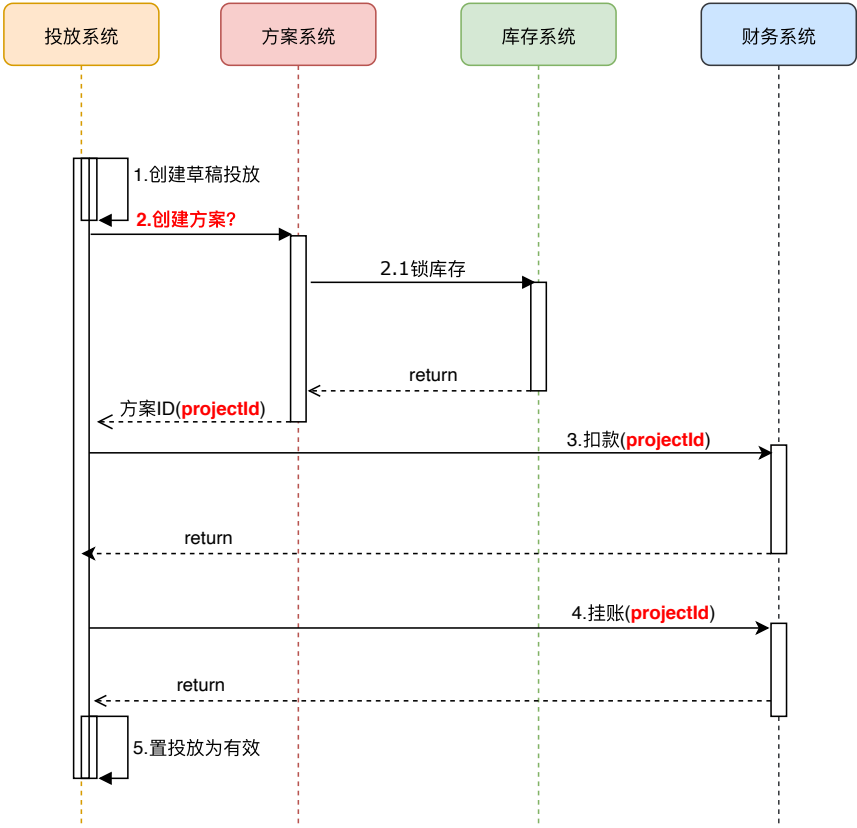
以公司分布式事务Swan的TCC为例

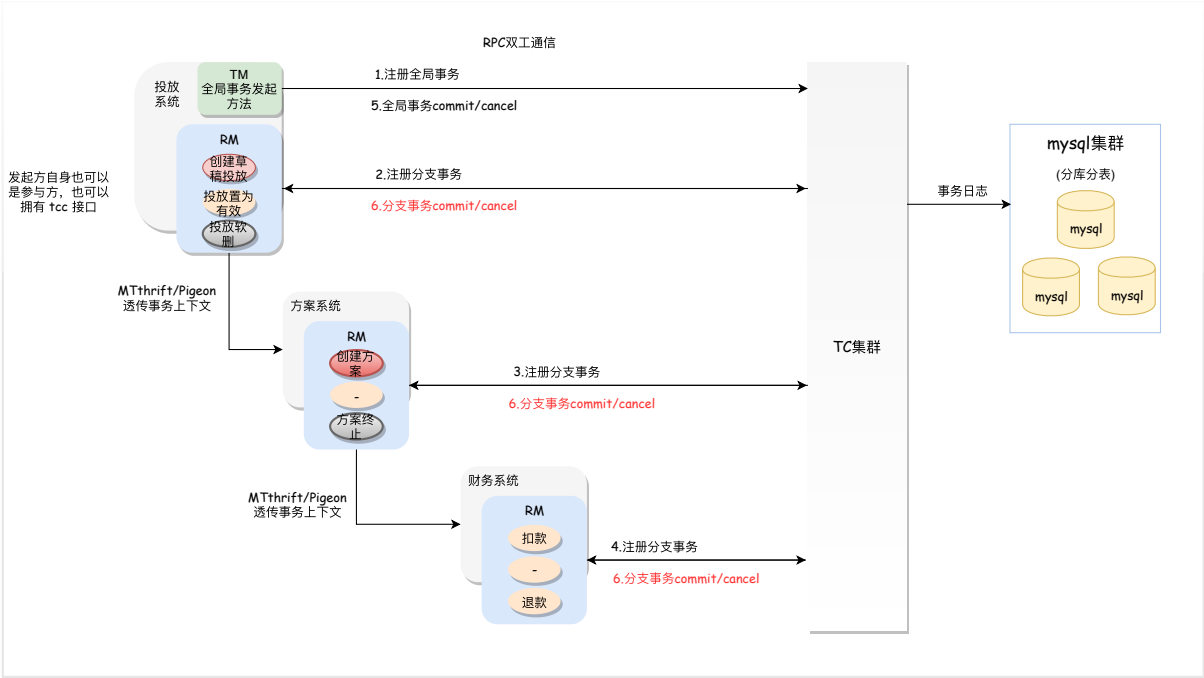


Swan TCC 模式的具有一个独立部署的事务协调 TC（事务协调器，Transaction Coordinator）集群。业务在发起方发起一个分布式并以此调用下游服务后，会向 TC 集群 提交事务执行结果，TC 根据执行结果判断对该分布式事务进行 confirm(cancel)。

(3) 思考

想到一个咱们这边存在分布式事务的场景，就是结婚CPT创建投放的第一版流程（我们自己实际上有实现分布式事务，不过是基于消息的回补，这种模式也被称为SAGA模式）





(4) 问题

TCC并不是完美的，也存在一些问题：

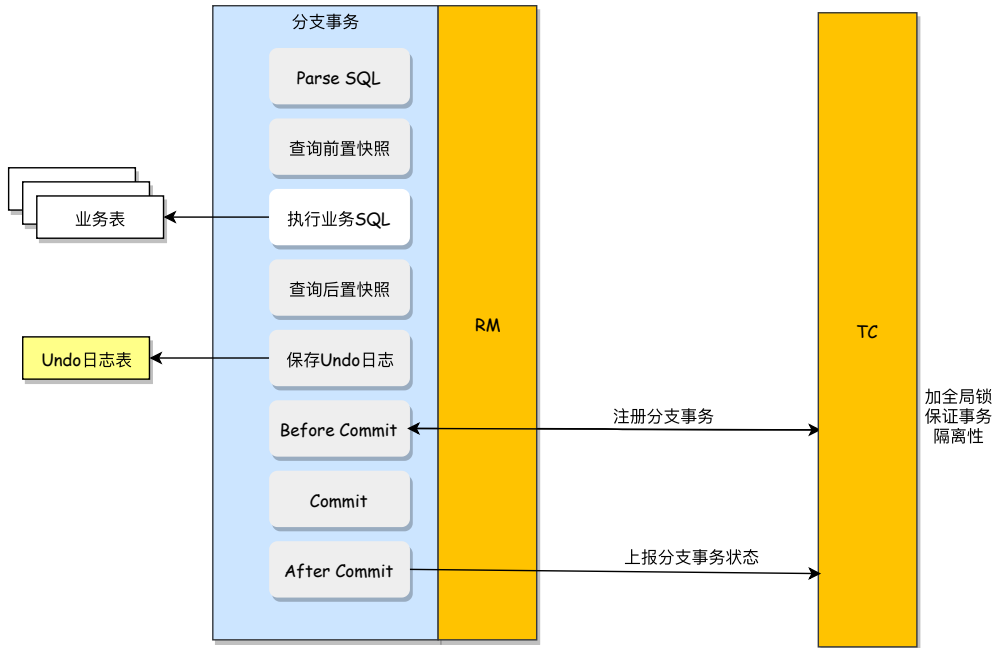
- 对于一个原本不支持TCC的系统，需要考虑将业务拆解为两个部分，即资源预留（Try），确认（Confirm）两阶段
- 对业务系统侵入太深，需要事务参与各方配合改造，最起码事务参与方都得实现三个接口

2.3.2 MTXC/Seata

相比XA和TCC，始终都有各自的不好，比如XA强一致性带来的可用性差，而TCC呢，对业务侵入太深（上游下游一起改代码！！），那么它来了，阿里的开源产品Seata，美团Swan推出了MTXC，是应用层面的**两阶段提交协议(2PC)**，属于柔性补偿型事务，具有代码无侵入，接入简单，适用场景广泛等优点。广泛应用于分库分表事务和跨库事务

执行流程

阶段1:



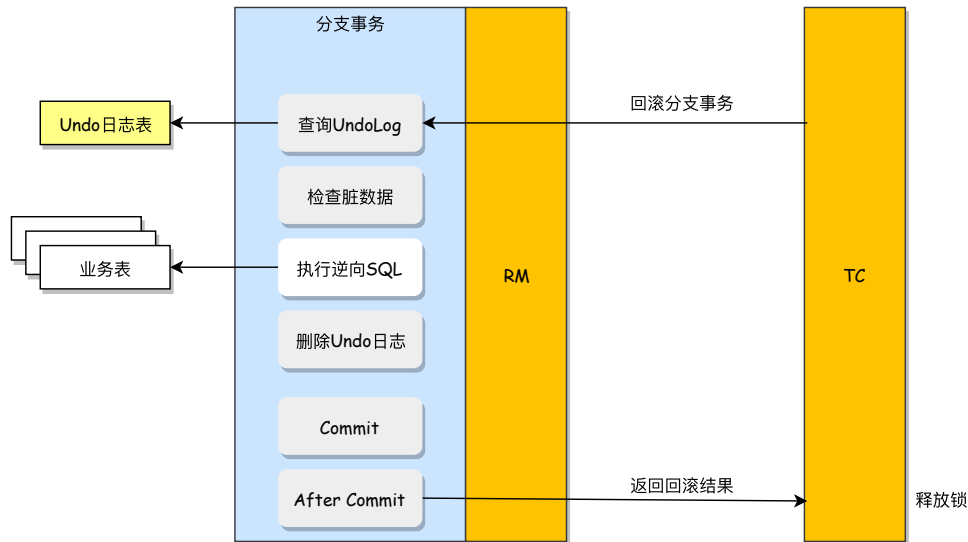
阶段1主要为生成undolog并上报事务信息，在TC加全局锁。

阶段2:
分支提交



阶段2 如果执行成功，则仅需异步删除undolog

阶段2:
分支回滚



阶段2 如果需要进行回滚，则会从undolog中找到逆向sql恢复数据，然后删除undo日志上报结果。

说明:

- MTXC实际是通过在执行业务一阶段业务之前，先做数据快照，解析业务SQL并执行，会记录下执行生成的Undo日志（便于回滚）

- 当各事务分支第一阶段执行成功了，则在第二阶段删除undo日志
- 当存在事务分支第一阶段失败，则对已在第一阶段执行成功的事务分支，通过第一阶段的undo日志逆向生成回滚SQL，进行Cancel

2.3.3 SAGA

用Saga来实现跨越多个服务的业务事务。Saga代表着一系列本地事务。每个本地事务更新数据库，**发布消息或事件**来触发Saga里的下一个本地事务。如果一个本地事务由于违反了业务规则而失败，Saga执行一系列补偿事务以撤回被前面事务产生的更改。

有两种方式可以协调Saga：

- Choreography - 每个本地事务发起领域事件，触发其他服务里的本地事务
- Orchestration - 协调者对象告诉参与者执行哪个本地事务

参考

高性能MySQL

MySQL技术内幕：InnoDB存储引擎

Taobao数据库内核月报：<http://mysql.taobao.org/monthly/>

大神博客：<https://blog.jcole.us/innodb/>

非官方优化文：<http://www.unofficialmysqlguide.com/introduction.html>

BASE理论：<https://queue.acm.org/detail.cfm?id=1394128>

DTP模型：<http://pubs.opengroup.org/onlinepubs/9294999599/toc.pdf>

XA规范官方文档：<http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf>

🔒 仅供内部使用，未经授权，切勿外传