

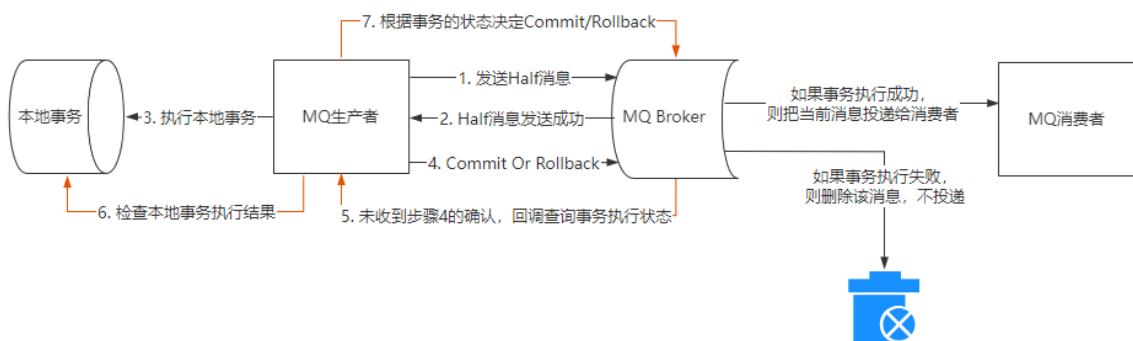
基于RocketMQ的事务消息方案

前面的内容中我们已经对RocketMQ有一个基本的了解。

RocketMQ是阿里巴巴开源的分布式消息中间件，于 2012 年开源，并在 2017 年正式成为 Apache 顶级项目。

Apache RocketMQ4.3版本之后，开始支持事务消息，事务消息为分布式事务提供了非常方便的解决方案，Apache RocketMQ4.3 整体实现流程如下图所示。

1. MQ生产者发送事务消息到MQ Broker中，此时MQ Broker把消息状态标记为(Prepared)，此时这条消息，MQ消费者是无法消费的。
2. MQ Broker回应MQ生产者，消息发送成功，表示MQ Broker已经接收到了消息并且保存成功。
3. MQ 生产者此时开始执行本地事务
4. MQ 生产者本地事务执行成功后自动向MQ Broker发送一个Commit消息，MQ Broker收到该消息后把第一步发送的那条消息标记为“可消费”，此时MQ 消费者可以正常收到这条消息进行处理。
5. MQ消费者消费完这条消息后，向MQ Broker发送一个ACK表示成功消费，否则MQ Broker会不断重试。
6. 如果MQ 生产者执行本地事务的过程中，MQ生产者宕机或者一直没有发送commit给到MQ Broker，MQ Server将会不停的询问同组的其他 Producer来获取事务执行状态。MQ Server会根据事务回查结果来决定是否投递消息



RocketMQ事务消息的设计，主要是解决Producer端发送消息和本地事务执行结果的原子性问题，因此RocketMQ的设计中Broker和Producer端提供了双向通信的能力，使得Broker天生可以作为一个事务协调者。

而RocketMQ本身提供的存储机制，为事务消息提供了持久化的能力。

再加上RocketMQ的高可用机制以及可靠性消息设计能力，为事务消息在系统发生异常时仍然能够保证消息的成功投递。

因此它的实现思想其实就是本地消息表的实现思路，只不过本地消息表移动到了MQ的内部，最终解决Producer端的消息发送和本地事务的原子性问题。

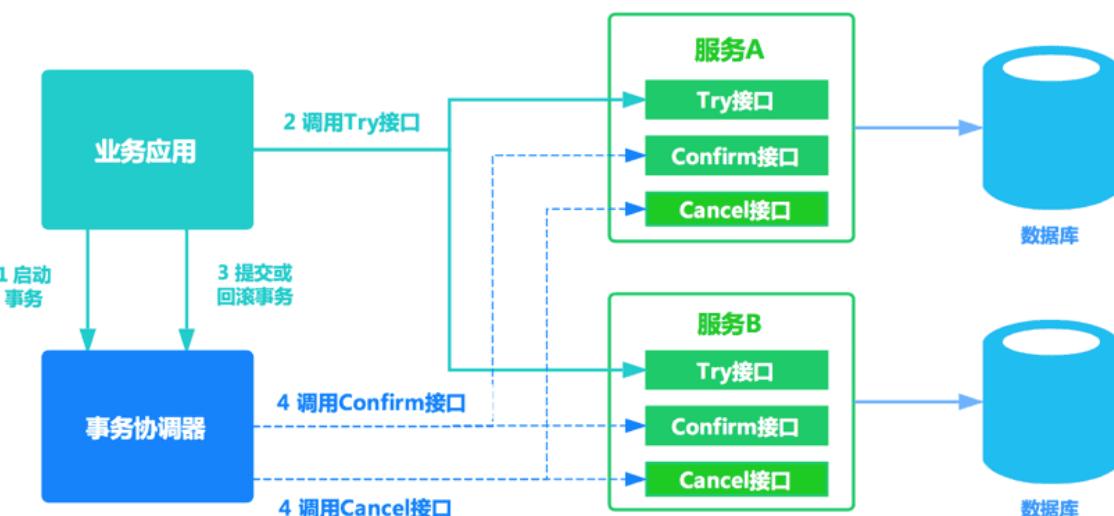
TCC事务解决方案

国内最早关于TCC的报道，应该是InfoQ上对阿里程立的一篇采访。经过程博士的这一次传道之后，TCC在国内逐渐被大家广为了解并接受。相应的实现方案和开源框架也先后被发布出来，ByteTCC就是其中之一；

TCC的方案，在电商、金融领域落地也比较多，他是一种两阶段提交的基于应用层的改进方案。TCC将整个业务逻辑的每个分支分成了Try、Confirm、Cancel三个操作，try部分完成业务的准备工作，confirm部分完成业务的提交、cancel部分完成事务的回滚

TCC事务解决方案本质上是一种补偿的思路，它把事务运行过程分成Try、Confirm/cancel两个阶段，每个阶段由业务代码控制，这样事务的锁力度可以完全自由控制。

需要注意的是，TCC事务和2pc的思想类似，但并不是2pc的实现，TCC不再是两阶段提交，而只是它对事务的提交/回滚是通过执行一段confirm/cancel业务逻辑来实现，并且也并没有全局事务来把控整个事务逻辑。



Seata简介

Seata 是一款开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务。

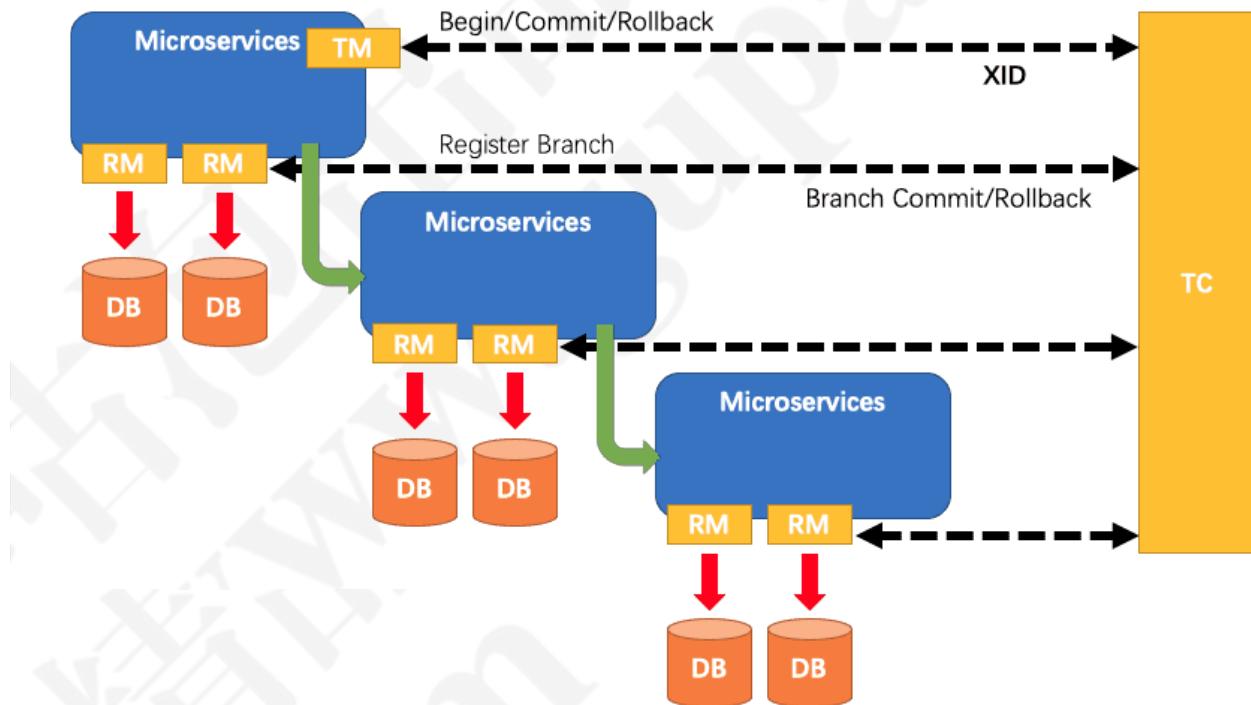
在 Seata 开源之前，Seata 对应的内部版本在阿里经济体内部一直扮演着分布式一致性中间件的角色，帮助经济体平稳的度过历年的双11，对各BU业务进行了有力的支撑。

经过多年沉淀与积累，商业化产品先后在阿里云、金融云进行售卖。2019.1 为了打造更加完善的技术生态和普惠技术成果，Seata 正式宣布对外开源，未来 Seata 将以社区共建的形式帮助其技术更加可靠与完备。

Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案。

Seata AT模式

Seata AT模式实际上是2PC协议的一种演变方式，也是通过两个阶段的提交或者回滚来保证多节点事务的一致性，它的工作模型如下图所示。



具体工作流程说明如下：

1. 第一阶段，应用系统会把一个业务数据的事务操作和回滚日志记录在同一个本地事务中提交，在提交之前，会向TC (seata server) 注册事务分支，并申请针对本次事务操作的表的全局锁。
接着提交本地事务，本地事务会提交业务数据的事务操作以及UNDO LOG，放在一个事务中提交。
2. 第二个阶段，这一个阶段会根据参与到同一个XID下所有事务分支在第一个阶段的执行结果来决定事务的提交或者回滚，这个回滚或者提交是TC来决定的，它会告诉当前XID下的所有事务分支，提交或者回滚。
 - 如果是提交，则把提交请求放入到一个异步任务队列，并且马上返回提交成功给到TC，这样可以避免阻塞问题。而这个异步任务，只需要删除UNDO LOG就行，因为原本的事务已经提交了。
 - 如果是回滚，则开启一个本地事务，执行以下操作
 - 通过XID和Branch ID查找到响应的UNDO LOG记录
 - 数据校验，拿到UNDO LOG中after image (修改之后的数据) 和当前数据进行比较，如果有不同，说明数据被当前全局事务之外的动作做了修改，这种情况需要根据配置策略来做处理。
 - 根据UNDO LOG中的before image和业务SQL的相关信息生成并执行回滚语句
 - 提交本地事务，并把本地事务的执行结果上报给TC

Seata AT模式原理详解

我们来通过一个简单的示例来说明Seata中AT模式的工作原理。

假设存在一个业务表 **Product**，表结构如下：

Field	Type	Key
id	bigint(20)	PRI
name	varchar(100)	
since	varchar(100)	

这个表中，有一条对应的数据：

id	name	since
1	TXC	2014

假设AT模式下的其中一个分支事务的执行业务逻辑对应的sql语句如下：

```
update product set name = 'GTS' where name = 'TXC';
```

在Seata的AT模式中，执行的过程如下

AT模式第一阶段

这个阶段主要分为两个过程。

1. 执行数据的修改
2. 记录回滚日志

具体过程如下：

1. 解析前面的Update语句，得到SQL类型（UPDATE）、表（Product）、条件（where name = 'TXC'）等相关信息。
2. 根据解析到的条件信息，生成一条查询语句用来查询修改之前的数据状态

```
select id , name , since from product where name = 'TXC'
```

得到如下结果：

id	name	since
1	TXC	2014

3. 执行上述的业务SQL，也就是更新name为GTC。
4. 在根据第二个步骤查询的主键id定位修改后的数据

```
select id, name, since from product where id = 1;
```

得到的结果如下：

id	name	since
1	GTS	2014

5. 有了更新前后的数据，以及业务SQL有关信息，组成一条回滚日志记录，插入到UNDO_LOG表中。

```

{
    "branchId": 641789253,
    "undoItems": [
        {
            "afterImage": {
                "rows": [
                    {
                        "fields": [
                            {
                                "name": "id",
                                "type": 4,
                                "value": 1
                            },
                            {
                                "name": "name",
                                "type": 12,
                                "value": "GTS"
                            },
                            {
                                "name": "since",
                                "type": 12,
                                "value": "2014"
                            }
                        ]
                    },
                    {
                        "tableName": "product"
                    }
                ],
                "beforeImage": {
                    "rows": [
                        {
                            "fields": [
                                {
                                    "name": "id",
                                    "type": 4,
                                    "value": 1
                                },
                                {
                                    "name": "name",
                                    "type": 12,
                                    "value": "TXC"
                                },
                                {
                                    "name": "since",
                                    "type": 12,
                                    "value": "2014"
                                }
                            ]
                        },
                        {
                            "tableName": "product"
                        }
                    ],
                    "sqlType": "UPDATE"
                }
            },
            "xid": "xid:xxx"
        }
    ]
}

```

6. UNDO日志提交之前，会向TC (SEATA-SERVER) 注册分支事务，并申请 **product** 表中，主键值为1记录的全局锁。
7. 业务数据的更新语句 (UPDATE) 和UNDO LOG一起提交，并将本地事务的提交结果上报到 TC。

AT模式第二阶段

这个阶段，主要是TC会根据各个分支事务的执行结果，来决定事务的提交或者回滚。

1. 如果所有分支事务的执行结果都正常，则提交事务。由于实际上各个本地事务在第一阶段已经提交了，所以只需要异步去删除当前事务分支对应UNDO LOG表中的记录即可。
2. 如果存在部分事务分支执行异常的情况，则需要对事务进行回滚，回滚步骤如下
 - 收到TC的分支回滚请求，事务参与者开启一个本地事务。

- 通过XID和BranchID查找到UNDO LOG中对应的记录
- 拿到数据后，先对数据进行校验，使用UNDO LOG中 **afterImages**（修改后的数据）和当前 **product** 表中的数据进行比较，如果发现数据不相同，说明数据被当前全局事务之外的程序修改过，这种情况需要根据配置的策略来进行处理
- 根据UNDO LOG中的 **beforeImages**（修改之前的数据）和业务SQL相关信息生成回滚语句并执行。

```
update product set name = 'TBC' where id = 1;
```

3. 提交本地事务，并把本地事务的执行结果（分支事务的回滚结果）上报给TC。

以上就是Seata AT事务模型的执行流程，其实从整体的实现思路上，类似于把数据库的事务在作用范围内做了一层升华，核心步骤和SQL类似：

1. 加锁
2. 写事务日志
3. 提交事务或回滚事务

在这种事务模型下，它的事务隔离级别是如何实现的呢？

Seata AT模式的事务隔离级别

我们在学习数据库的事务特性时，必须会涉及到的就是事务的隔离级别，不同的隔离级别，会产生一些并发性的问题，比如

- 脏读
- 不可重复读
- 幻读

我们知道mysql的数据库隔离级别有4种，具体哪四种我们就不在这里说明，后续在讲mysql的时候会讲到。

写隔离

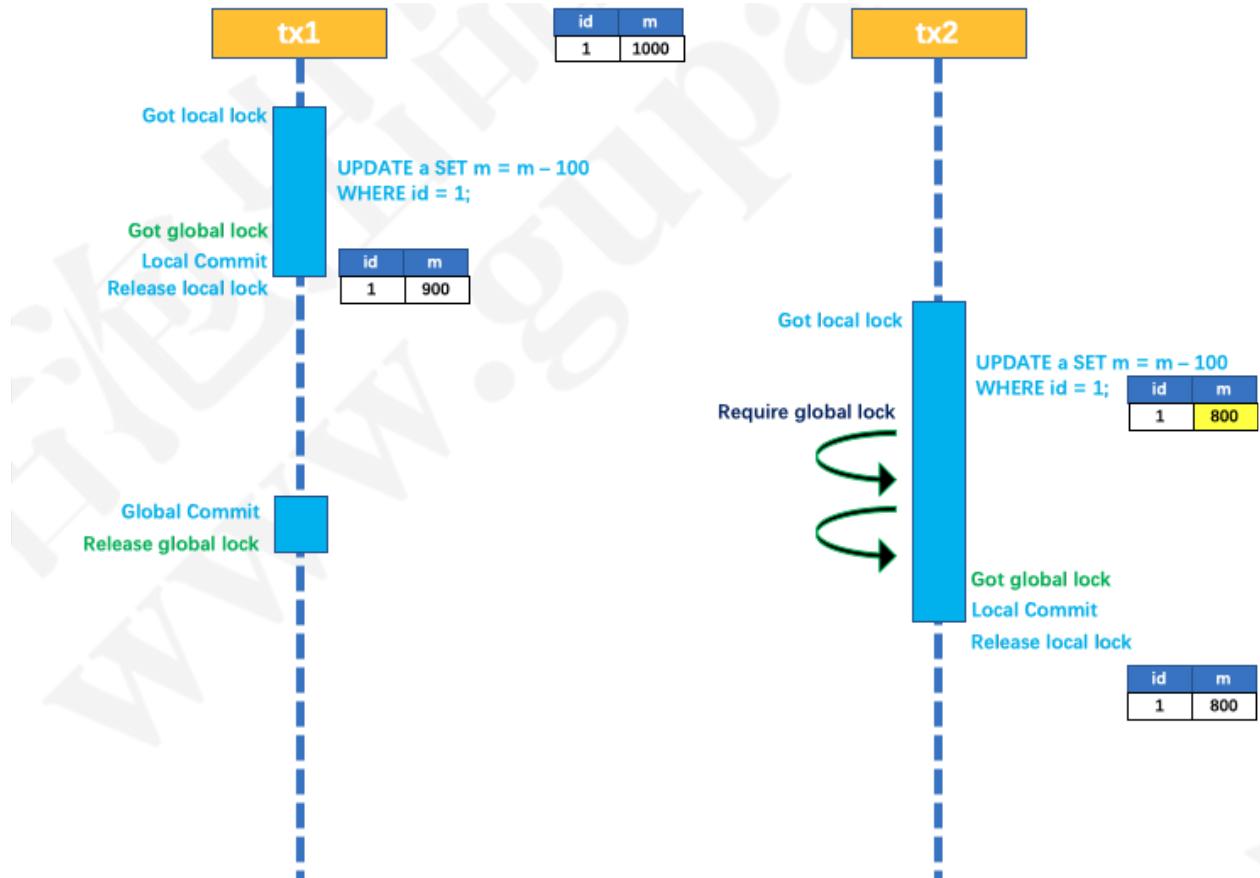
所谓的写隔离，就是多个事务对同一个表的同一条数据做修改的时候，需要保证对于这个数据更新操作的隔离性，在传统事务模型中，我们一般是采用锁的方式来实现。

那么在分布式事务中，如果存在多个全局事务对于同一个数据进行修改，为了保证写操作的隔离，也需要通过一种方式来实现隔离性，自然也是用到锁的方法，具体来说。

- 在第一阶段本地事务提交之前，需要确保先拿到全局锁，如果拿不到全局锁，则不能提交本地事务
- 拿到全局锁的尝试会被限制在一定范围内，超出范围会被放弃并回滚本地事务并释放本地锁。

举一个具体的例子，假设有两个全局事务tx1和tx2，分别对a表的m字段进行数据更新操作，m的初始值是1000。

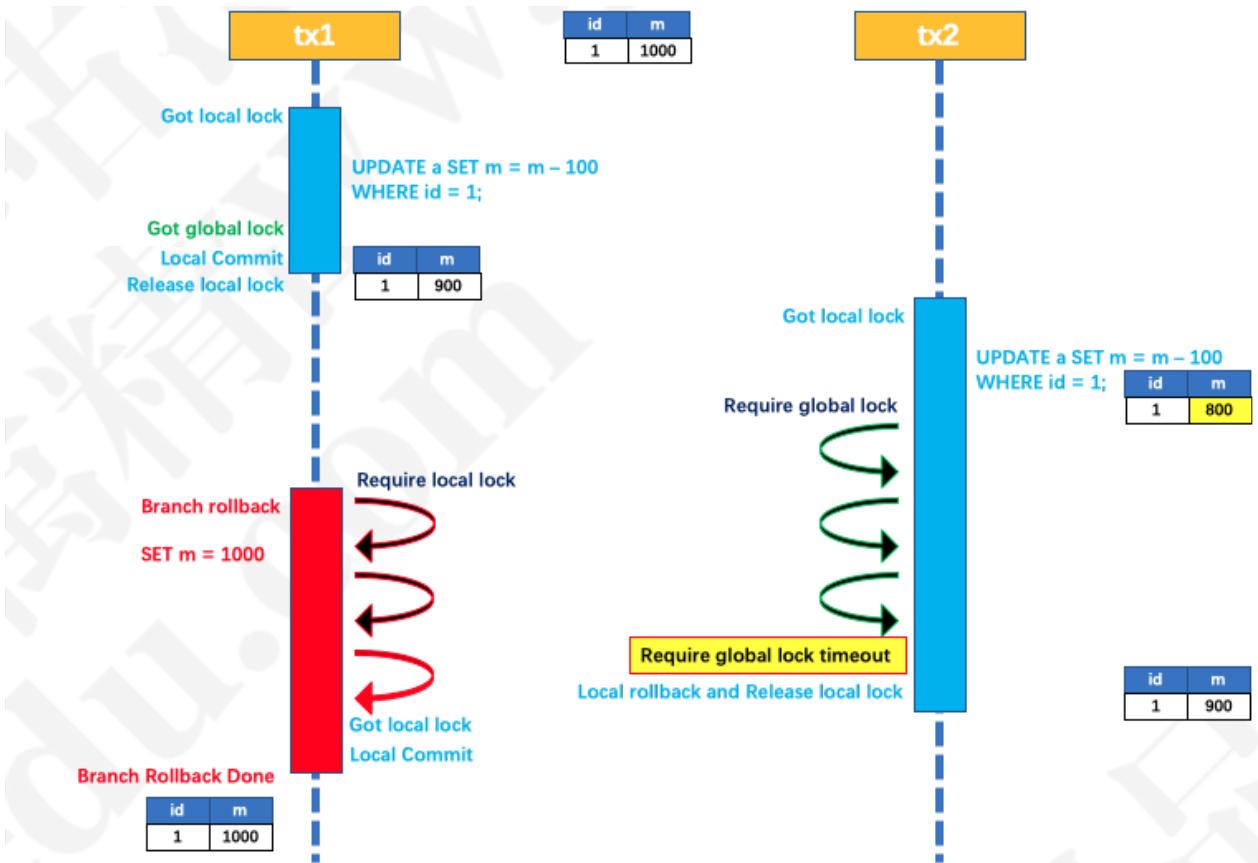
- tx1先开始执行，按照AT模式的流程，先开启本地事务，然后更新 $m=1000-100=900$ 。在本地事务更新之前，需要拿到这个记录的全局锁。
- 如果tx1拿到了全局锁，则提交本地事务并释放本地锁。
- 接着tx2后开始执行，同样先开启本地事务拿到本地锁，并执行 $m=900-100$ 的更新操作。在本地事务提交之前，先尝试去获取这个记录的全局锁。而此时tx1全局事务还没提交之前，全局锁的持有者是tx1，所以tx2拿不到全局锁，需要等待



接着，tx1在第二阶段完成事务提交或者回滚，并释放全局锁。此时tx2就可以拿到全局锁来提交本地事务。当然这里需要注意的是，如果tx1的第二阶段是全局回滚，则tx1需要重新获取这个数据的本地锁，然后进行反向补偿更新实现事务分支的回滚。

此时，如果tx2仍然在等待这个数据的全局锁并且同时持有本地锁，那么tx1的分支事务回滚会失败，分支的回滚会一直重试直到tx2的全局锁等待超时，放弃全局锁并回滚本地事务并释放本地锁之后，tx1的分支事务才能最终回滚成功。

由于在整个过程中，全局锁在tx1结束之前一直被tx1持有，所以并不会发生脏写问题。



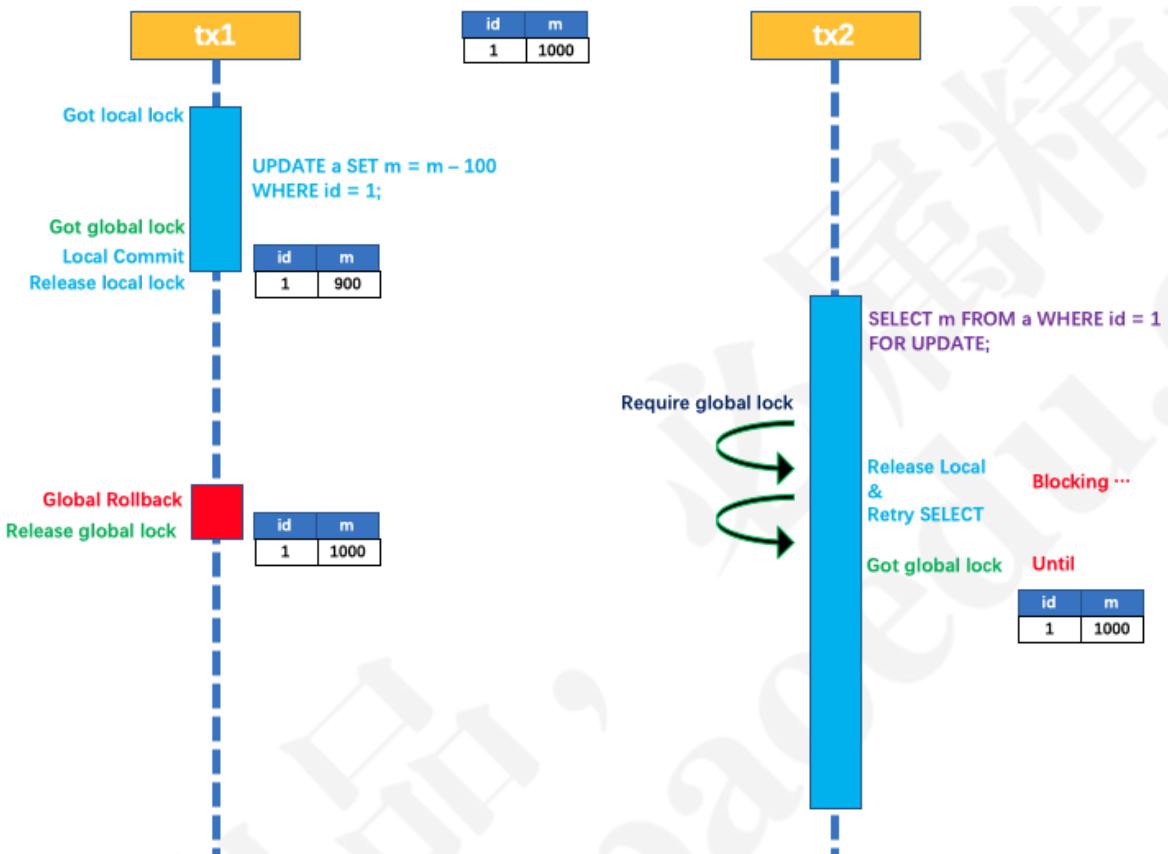
读隔离

在数据库本地事务隔离级别 **读已提交 (Read Committed)** 或以上的基础之上，Seata (AT 模式) 的默认全局隔离级别是 **读未提交 (Read Uncommitted)**。其实从前面的流程中就可以很显而易见的分析出来，因为本地事务提交之后，这个数据就对外可见，并不用等到tc触发全局事务的提交。

如果在特定场景下，必须要求全局的读已提交，目前Seata的方式只能通过SELECT FOR UPDATE语句来实现。

SELECT FOR UPDATE 语句的执行会申请 **全局锁**，如果 **全局锁** 被其他事务持有，则释放本地锁（回滚 SELECT FOR UPDATE 语句的本地执行）并重试。这个过程中，查询是被 block 住的，直到 **全局锁** 拿到，即读取的相关数据是 **已提交** 的，才返回。

出于总体性能上的考虑，Seata 目前的方案并没有对所有 SELECT 语句都进行代理，仅针对 FOR UPDATE 的 SELECT 语句。



应用实战

详见附件源码