Alibaba 分布式事务 seata 学习笔记

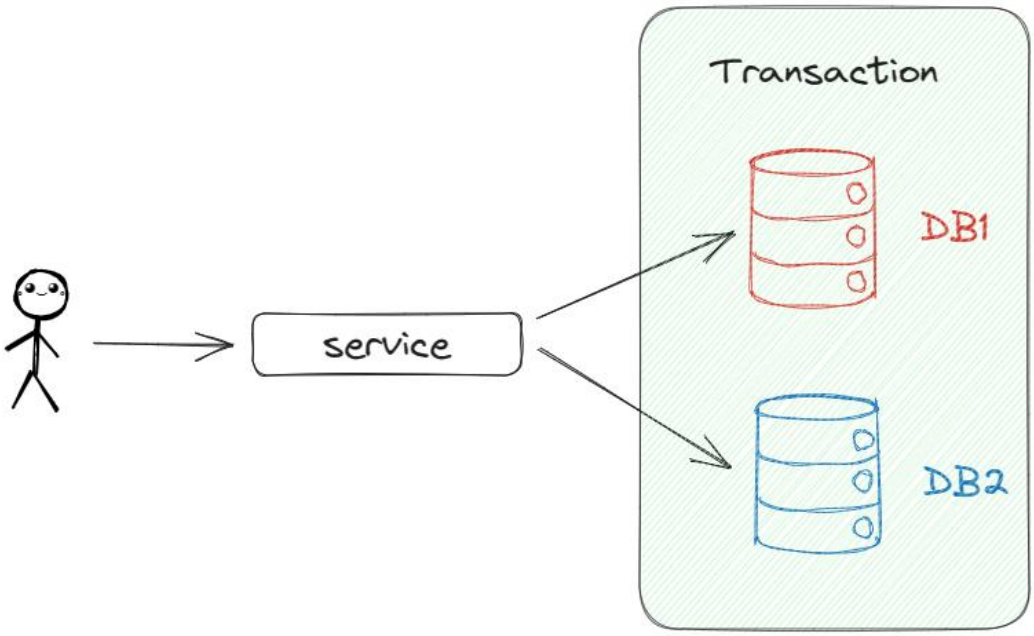
一、分布式事务概述

1．什么是分布式事务

一个操作单元，要么一起成功，要么一起失败，但是这个操作单元的不同操作位于不同的服务器上，称之为分布式事务。

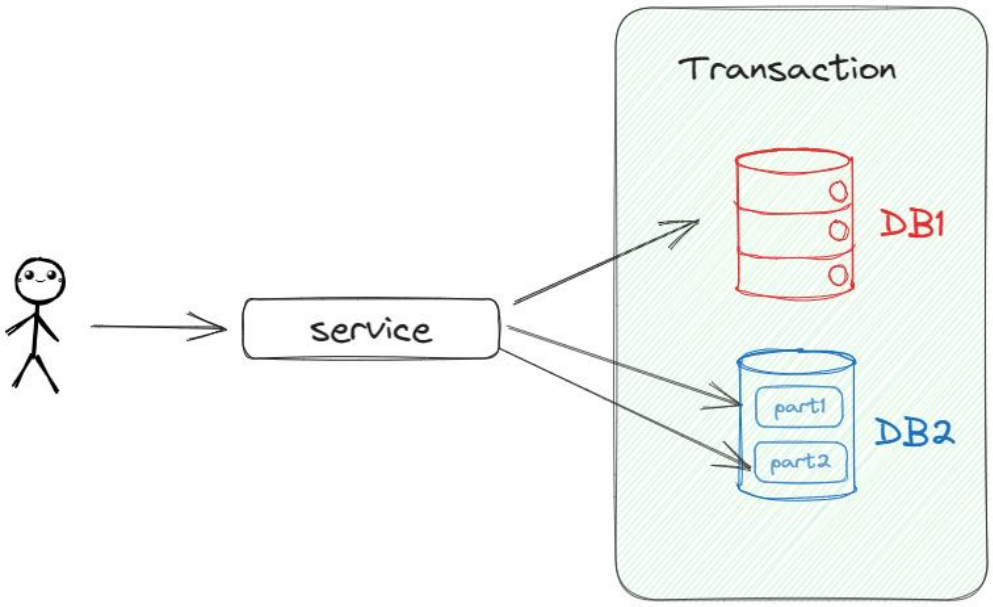
2．分布式事务的几种场景

（1）跨库事务--->多个数据库数据参与同一事务



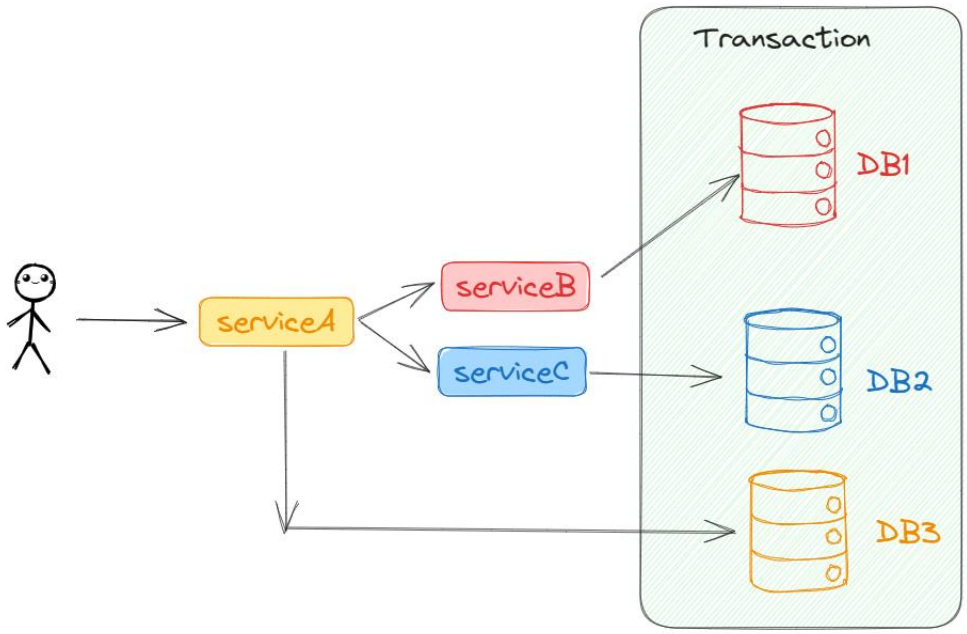
（2）分库分表

为了保证大数据量的查询速度，有的系统采取了分库分表的，对于分库分表的查询，就会涉及分库分表



（3）微服务化

多个微服务参与同一事务，并且涉及多个数据库的数据操作，考虑分布式事务

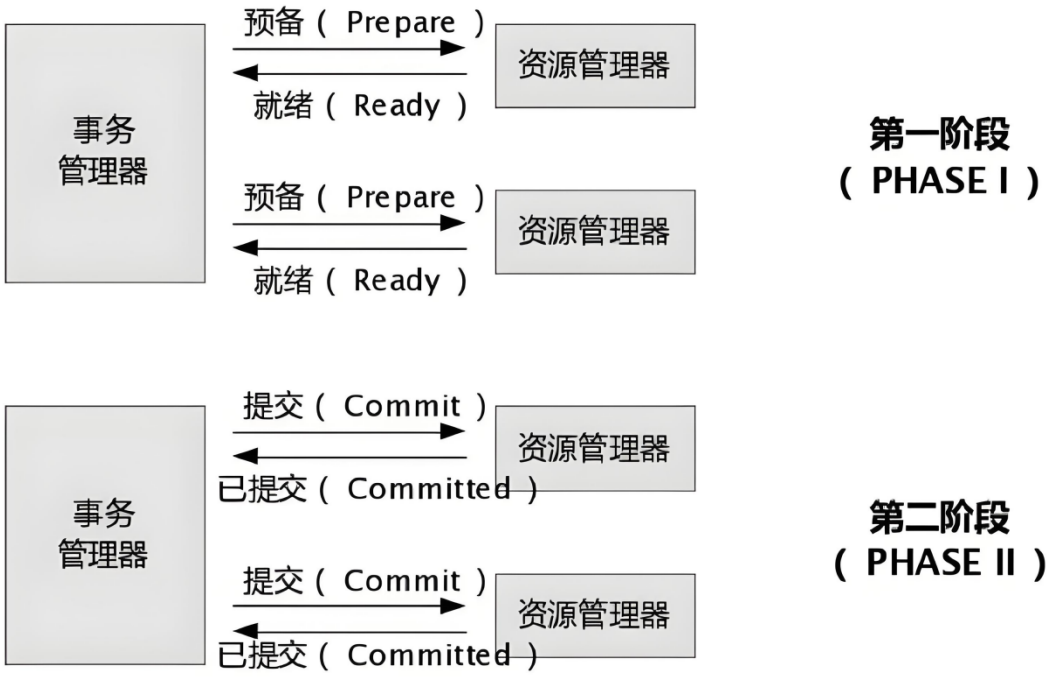


二、分布式事务的解决方案

1．2PC（两阶段提交）

（1）XA 协议

XA 协议是分布式事务协议，由事务管理器和多个资源管理器组成，XA 协议又叫两阶段提交方案，分为 prepare 和 commit 两个阶段；

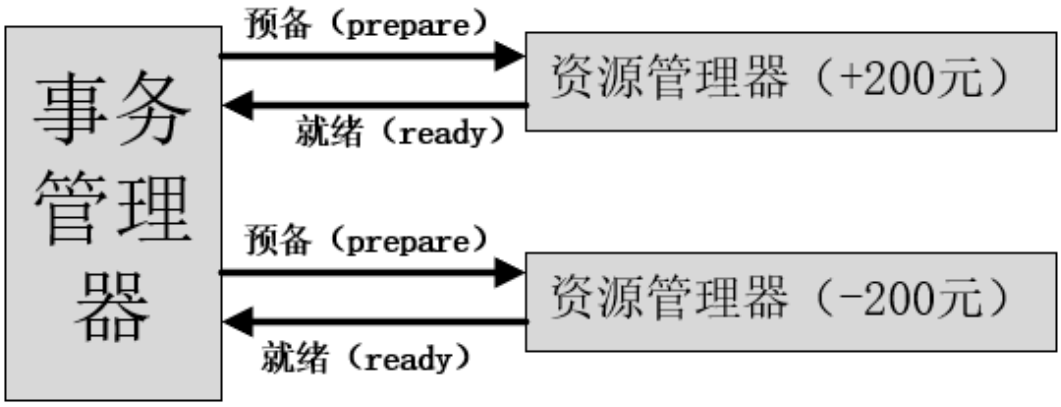


以转账为例：

① 第一阶段 Prepare 预备阶段

事务管理器会向参与分布式事务的数据操作，发送 prepare 指令，如果数据操作准备好，就会返回 ready 响应，表示数据操作已经就绪；

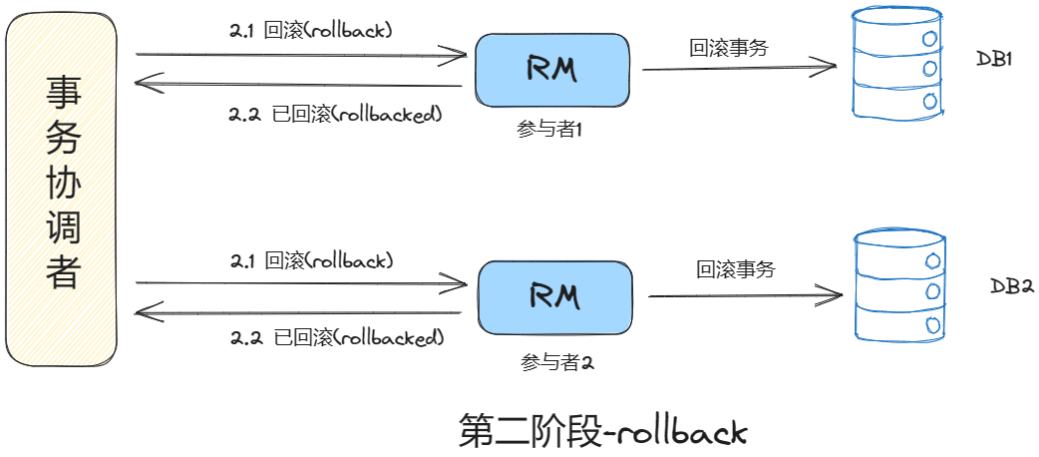
如果数据操作并没有全部返回 ready 响应，就会回滚 prepare 操作；



② 第二阶段 Commit 提交阶段

事务管理器会向参与分布式事务的数据操作，发送 commit 指令，如果数据操作都已经提交数据，就会返回 commited 响应，表示数据操作已经提交完成；

如果数据操作并没有全部返回 commited 响应，就会回滚 commit 操作；



③ 优缺点

- 优点：
  - 尽量保证了数据的强一致性，适合对数据强一致性要求比较高的领域；
- 缺点：
  - 性能问题：数据在整个事务过程中，被操作的数据是事务所涉及的线程所独享的，不能被第三方线程进行访问的。需要等待事务提交将锁释放后，第三方线程才能访问；
  - 可靠性问题：2PC 非常依赖资源的协调者，当协调者出现故障，尤其是第二阶段出现故障时，数据将永远处于锁定状态而无法被其他资源访问；
  - 一致性问题：当处于第二阶段，事务管理器向各个资源管理器发送回滚命令时，一部分资源管理器因为网络等原因，没有收到回滚命令，就会发生数据不一致问题；
  - 2PC 无法解决的问题：当协调者/事务管理器发送 commit 命令后宕机，接收到 commit 命令的资源管理器也发生了宕机；这时，即使事务管理器做了集群，选出了新的事务管理器，刚才提交的事务所处状态也无法确定；

2．3PC（三阶段提交）

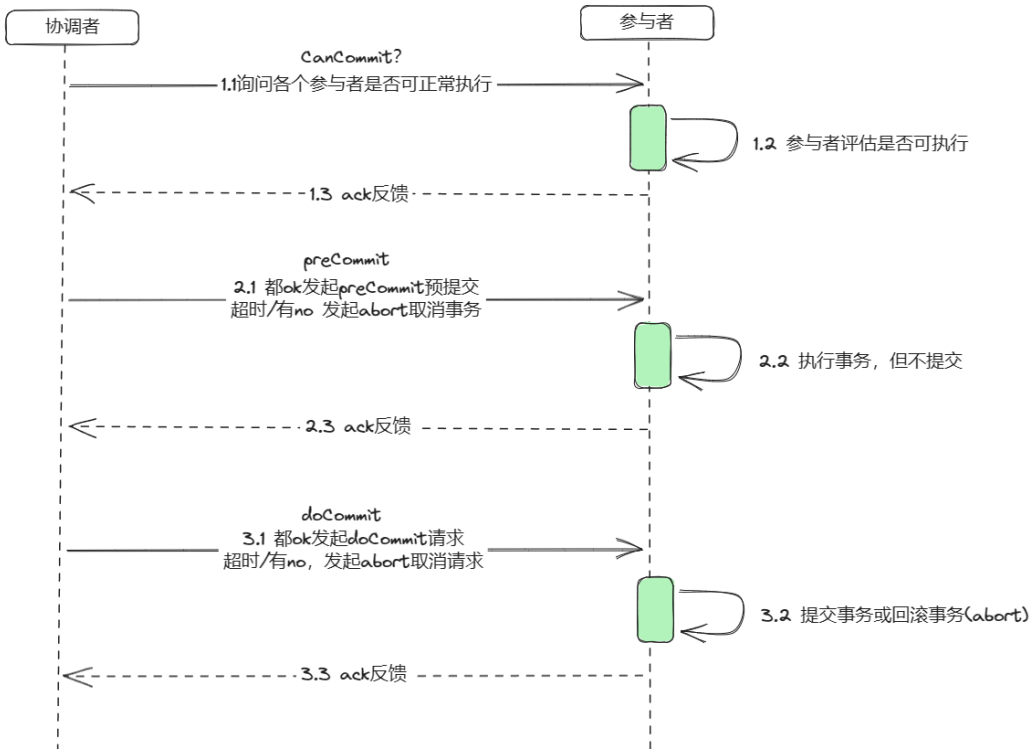
（1）概念

3PC 又叫三阶段提交，是针对 2PC 缺点的优化版，具体改进如下：

- 增加了协调者、参与者接受消息超时的机制；
- 增加了参与者的确认机制；

（2）概述

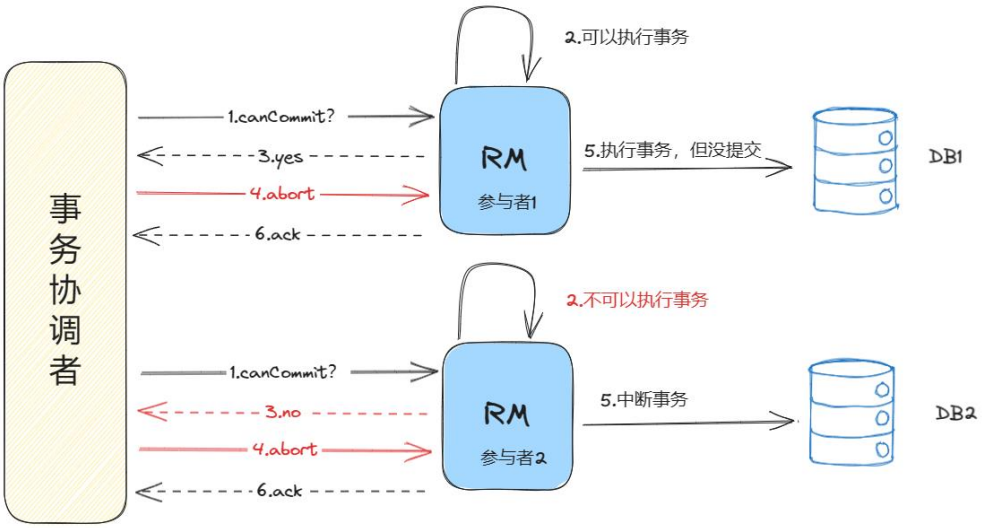
3PC 分为 3 个阶段：CanCommit 准备阶段、PreCommit 预提交阶段、DoCommit 提交阶段。



（3）工作流程总结

① 协调者向参与者发送 CanCommit 的命令，参与者进行自检，判断是否能够进行事务提交操作；

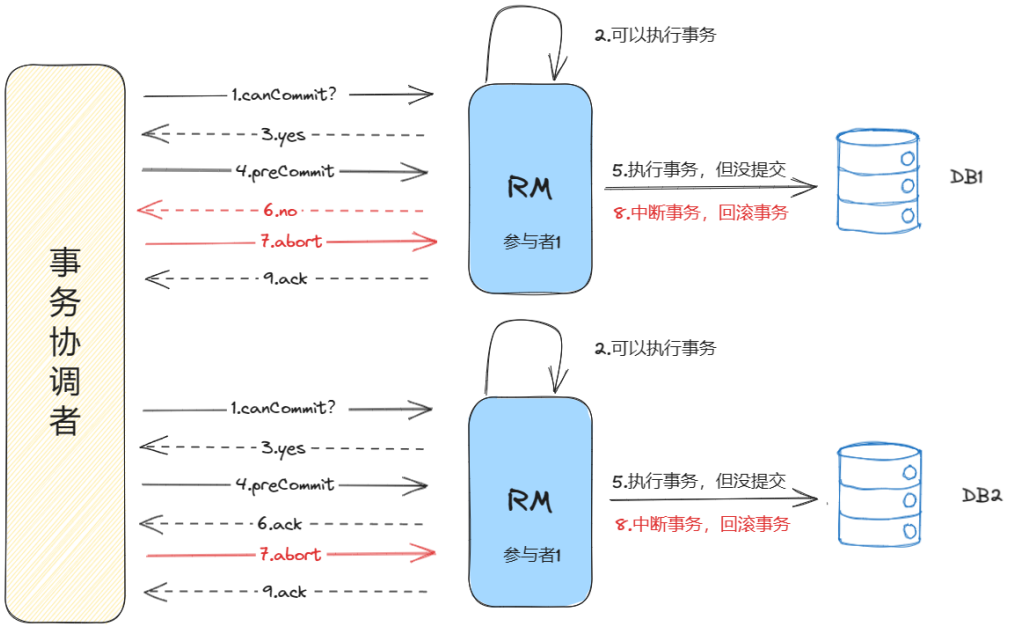
② 如果有参与者向协调者发送了 NO 响应，或者协调者等待时间过长未收到相应，协调者就会向所有参与者发送 abort 请求，来中断事务；或者参与者长时间未收到协调者的下一个指令，也会主动中断事务。



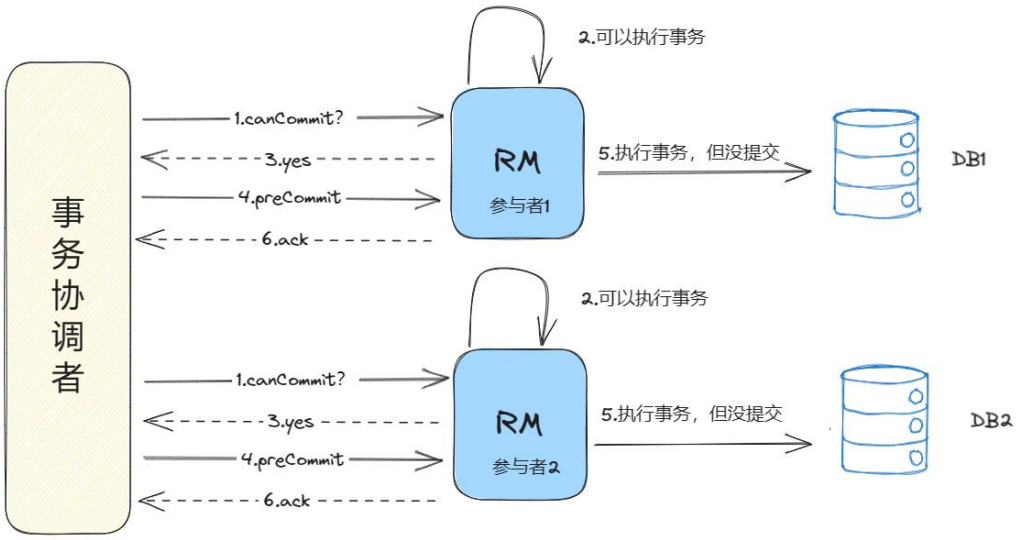
③ 如果参与者向协调者发送了 YSE 响应，就说明参与者现在有能力提交事务；协调者就会继续向参与者发送 PreCommit 命令，等待参与者响应；

④ 如果有的参与者不能预提交事务，或者协调者等待参与者响应时间过长，协调者就会向参与者发送 abort 命令，来回滚操作，并且中断事务，然后向协调者发送 ACK 响应，表示已经回滚并且中断事务；或者参与者长时间等不到协调者的下一个命令，参与者同样会

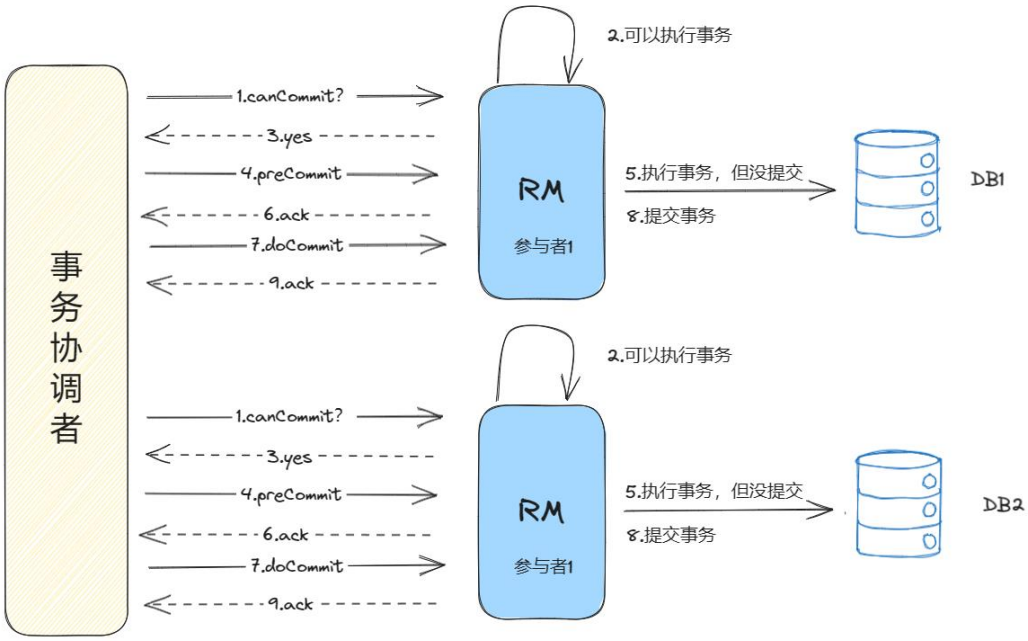
主动放弃事务；



⑤ 如果参与者都能执行事务预提交，那就执行事务，但是不提交，然后向协调者发送 ACK 响应，表示预提交完毕；



⑥ 在所有参与者完成了预提交操作，并返回 ACK 响应后，协调者就会向参与者发送 DoCommit 命令，进行事务的提交；参与者向协调者发送 ack 响应，协调者接收到所有参与者的 ack 响应之后，完成事务。



（4）优缺点

① 优点

- 协调者和参与者加入了超时机制，在长时间未接收到对方消息后，会回滚并放弃事务，防止因为网络，单点故障导致的问题；

② 缺点

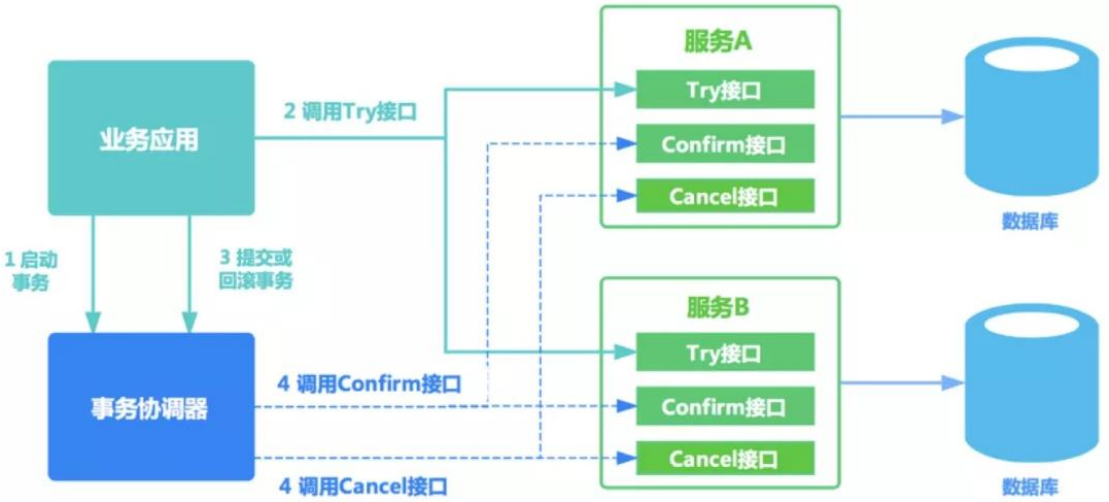
- 数据不一致问题依然存在，部分参与者在预提交完成后，等待提交的命令时，中断了和协调者的连接，协调者只能发送提交命令给一部分参与者，造成了数据不一致；

3．TCC

（1）概述

TCC 是一种分布式事务思想，需要用户自行编写代码完成，也是目前最火的分布式事务解决方案；

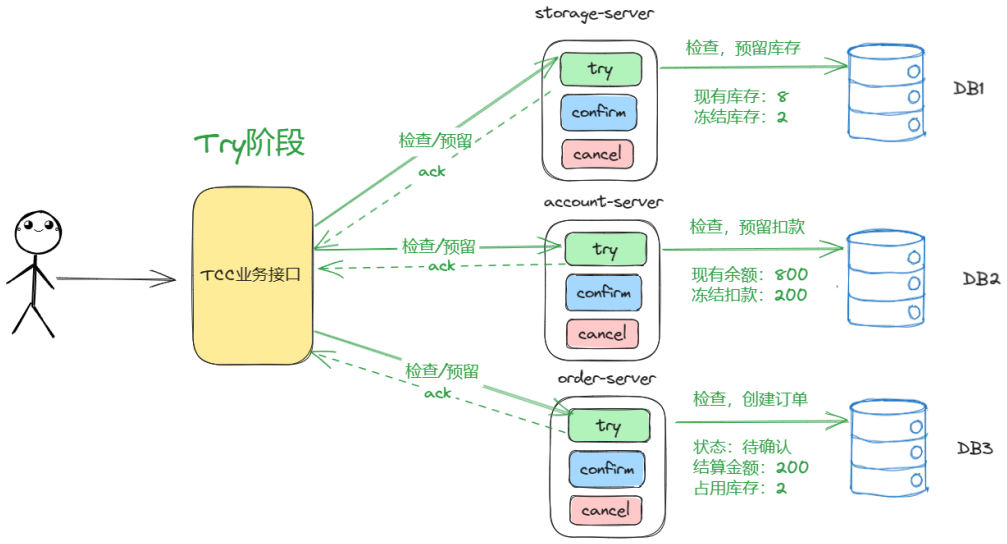
TCC 分为三个阶段 Try、Commit、cancel



（2）TCC 流程

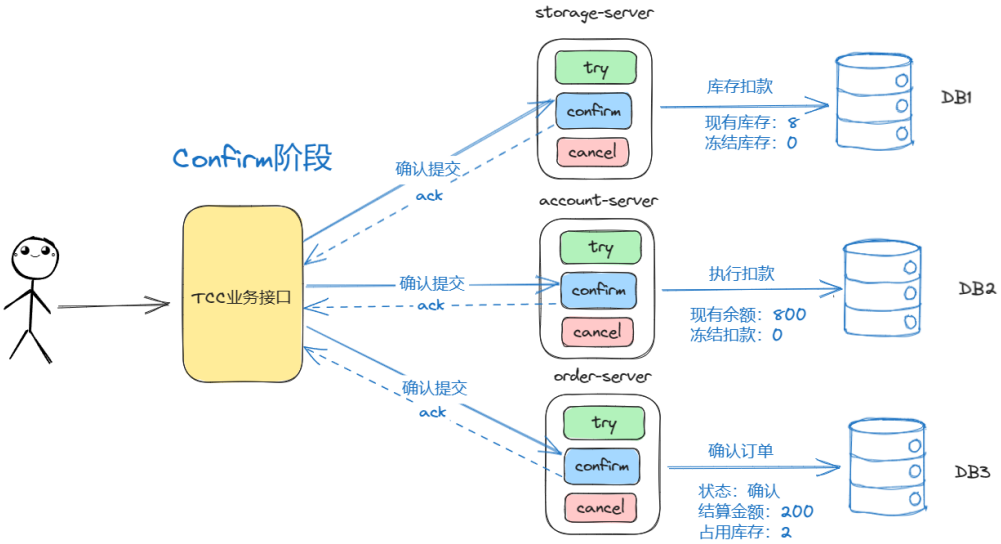
① Try 阶段

完成所有操作的预提交，并进行一致性检查，返回预提交结果



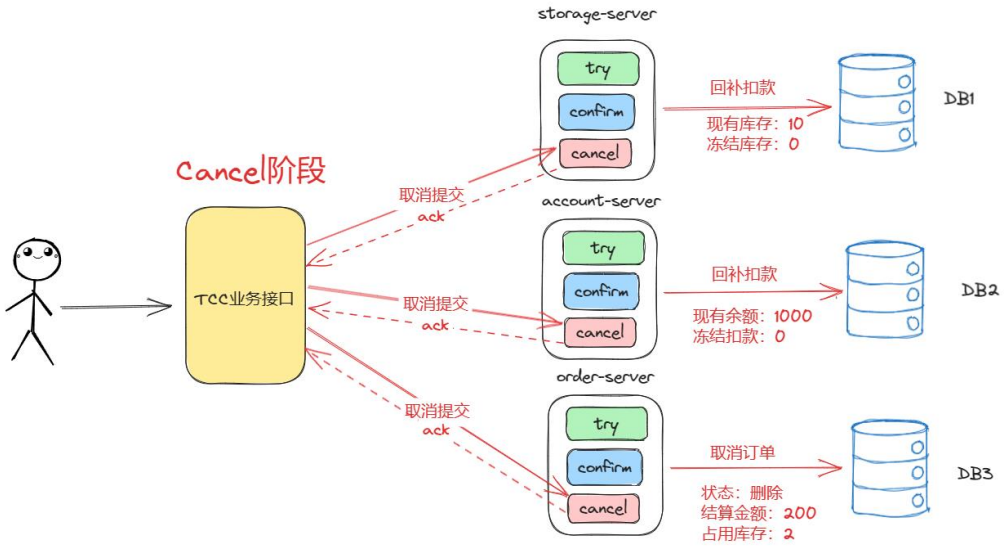
② Confirm 阶段

如果预提交结果完成，则进行数据的正式提交，TCC 认为 Confirm 阶段是不会出错的。即：只要 Try 成功，Confirm 一定成功。若 Confirm 阶段真的出错了，需引入重试机制或人工处理。



③ Cancel(取消)

如果预提交结果失败，则进行回滚，TCC 则认为 Cancel 阶段也是一定成功的。若 Cancel 阶段真的出错了，需引入重试机制或人工处理。



④ 注意要点

confirm 或者 cancel 有可能会重试，因此对应的部分需要支持幂等，防止重复提交或者重复回滚；

（3）优缺点

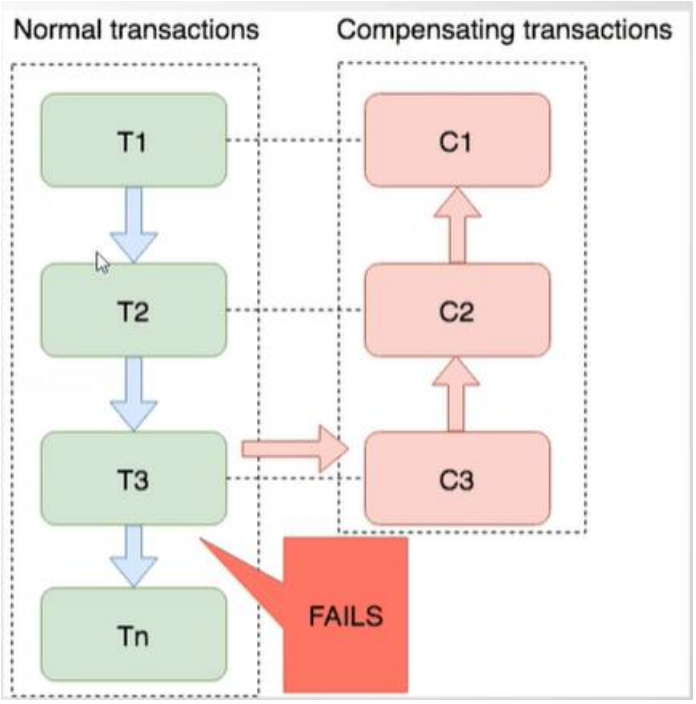
① 优点

- 业务代码可以做成集群，并且引入了重试机制，解决了 2PC、3PC 由于协调者单点故障引起的数据不一致问题，数据可以达到最终一致性。
- 数据锁的粒度变小，提高了性能。

② 缺点

- 提高了开发成本

4．SAGA（/ˈsɑːgə/）



（1） 概念

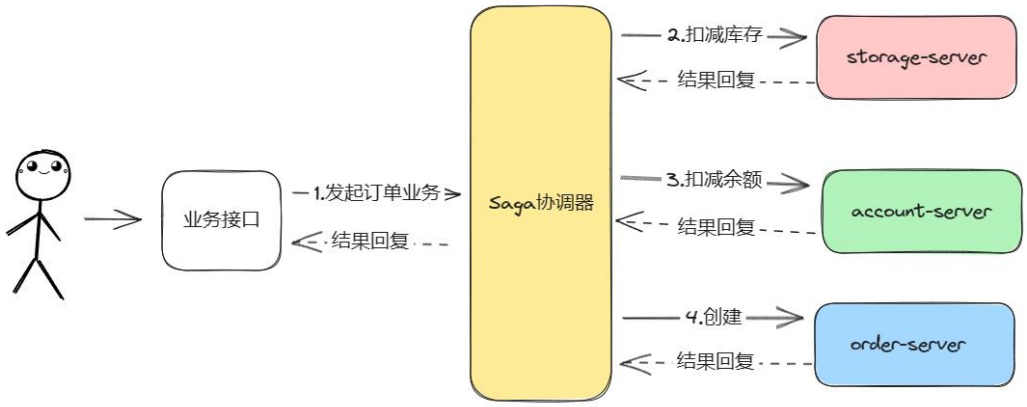
Saga 提供的是长事务、有第三方接口（如支付宝、微信支付的接口）参与解决方案，对于数据的操作通常需要两套代码，一套代码是用来正向操作，即数据的提交，另一套代码是数据恢复操作。实现有很多种方式，其中最流行的两种方式是：

- 命令协调
- 事件编排

（2）命令协调

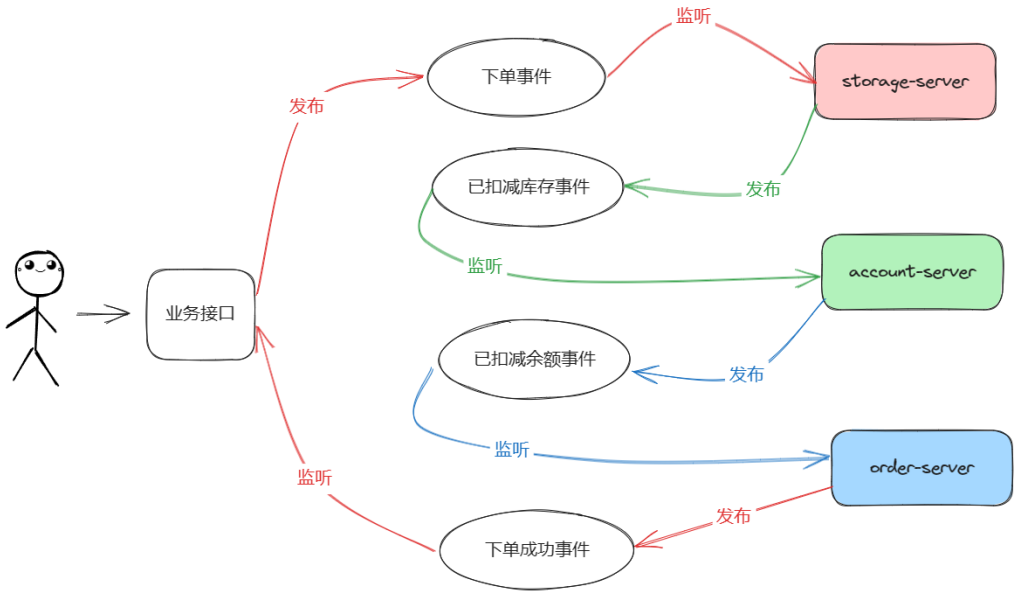
协调器以一定的顺序，以命令/回复的方式调用事务中的每一个操作，如果有一个操作回复异常，则以逆向的顺序进行回滚。

执行顺序： A-->B-->C　　回滚顺序： C-->B--->A



（3）事件编排

分布式事务以正常的顺序执行，每完成一次数据操作，就会监测下一次数据操作，如果出现异常，则执行回滚代码；



（4）Saga 的数据异常恢复

- 正向恢复：目的是分布式事务一定要成功，如加入重试机制，不停的重试；
- 反向恢复：即进行数据或者操作的回滚

（5）优缺点

① 命令协调模式

- 优点
  - 避免了循环依赖问题，更容易扩展
- 缺点
  - 存在单点故障问题

② 事件编排模式

- 优点
  - 避免了单点故障问题
- 缺点
  - 存在循环依赖问题

三、Seata 开发

1．简介

Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式,其中常用模式为 AT 和 TCC，默认为 AT

Seata 主要由三部分组成：事务协调者（TC），事务管理器（TM），资源管理器（RM）

	XA	AT	TCC	SAGA
一致性	强一致	弱一致	弱一致	最终一致
隔离性	完全隔离	基于全局锁隔离	基于资源预留隔离	无隔离
代码侵入	无	无	有，要编写三个接口	有，要编写状态机和补偿业务
性能	差	好	非常好	非常好
场景	对一致性、隔离性有高要求的业务	基于关系型数据库的大多数分布式事务场景都可以	• 对性能要求较高的事务。 • 有非关系型数据库要参与的事务。	• 业务流程长、业务流程多 • 参与者包含其它公司或遗留系统服务，无法提供 TCC 模式要求的三个接口

2． Seata XA 模式

（1）XA 模式概述

XA 模式是在 2PC 的 XA 协议基础上，进行适当的扩展和改进来实现的，增加了事务协调者和事务的注册机制；

（2）XA 模式的具体流程

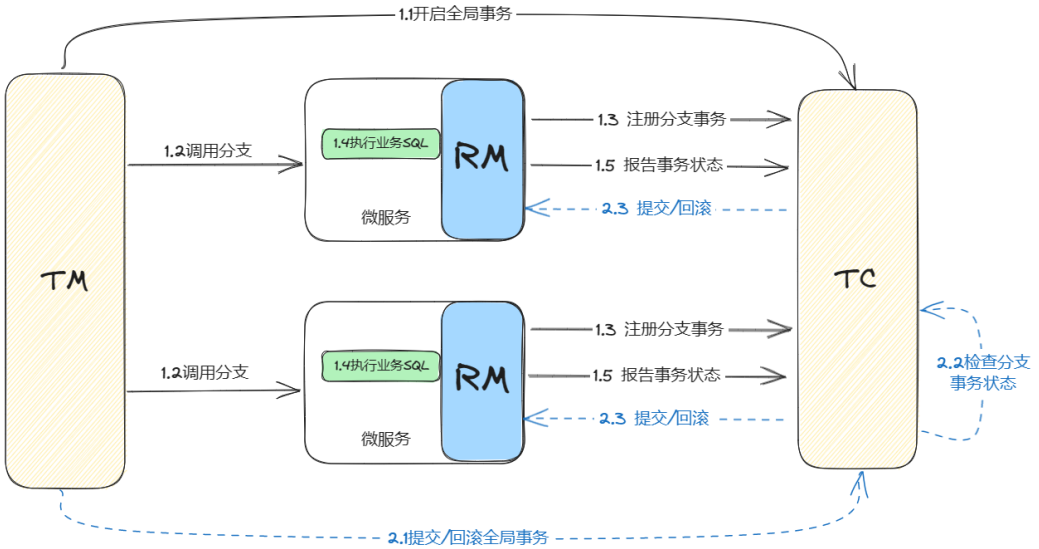
① 首先事务管理者会向事务协调者通知，并开启事务；

② 然后事务管理者会调用分布式事务的每一个数据操作；

③ 每个数据操作在执行本地事务前，都会通知事务协调者本地事务已经开始，并在本地事务预提交结束后，通知事务协调者本地事务执行的情况；

④ 在所有的数据操作都执行完毕后，事务管理者会向事务协调者发送提交/回滚操作通知；

⑤ 事务协调者会根据每一个本地事务的返回执行情况来通知每一个本地事务的提交和回滚；



（3）实现层面（具体详见 CSDN 或者视频教程）

① 除了必要的 seata 依赖配置后，只需要在 YML 的 seata 配置中设置分布式事务的模式为 XA

```yaml
#seata客户端配置
seata:
  enabled: true
  application-id: seata_tx
  tx-service-group: seata_tx_group
  service:
    vgroup-mapping:
      seata_tx_group: default
  registry:
    type: nacos
    nacos:
      application: seata-server
      server-addr: 127.0.0.1:8848
      namespace:
      group: SEATA_GROUP
  data-source-proxy-mode: XA
```

② 代码层面，只需要使用注解 @GlobalTransactional 来开启全局事务，每一个本地事务建议添加@Transactional,当然也可以不添加，并不影响全局事务的运行。

```java
@Override
@GlobalTransactional
public void purchase(String userId, String commodityCode, int orderCount, boolean rollback
    String result = stockFeignClient.deduct(commodityCode, orderCount);

    if (!"SUCCESS".equals(result)) {
        throw new RuntimeException("库存服务调用失败,事务回滚!");
    }
    result = orderFeignClient.create(userId, commodityCode, orderCount);
    if (!"SUCCESS".equals(result)) {
        throw new RuntimeException("订单服务调用失败,事务回滚!");
    }
```

（4） 优缺点

① 优点

- 事务的强一致性，每一个阶段都能保持一致

② 缺点

- 预提交阶段锁定数据库资源，提交阶段完毕后才释放，性能较差
- 依赖关系型数据库实现

3．Seata AT 模式

（1）AT 模式概述

AT 模式是分布式事务的默认模式；

AT 模式的本地事务默认隔离级别是读未提交，但是在特定的场景下要求是读已提交及以上；

在 XA 模式中，数据存在预提交和提交/回滚两个阶段，锁定数据时间较长，性能较差；

AT 模式可以看成是 XA 模式的优化版，在 AT 模式中，将 XA 模式的预提交阶段改为直接提交，并记录快照数据 undo log 日志，如果要回滚的话，就利用 undo log 日志进行回滚；

（2）Seata 的 undo log 介绍

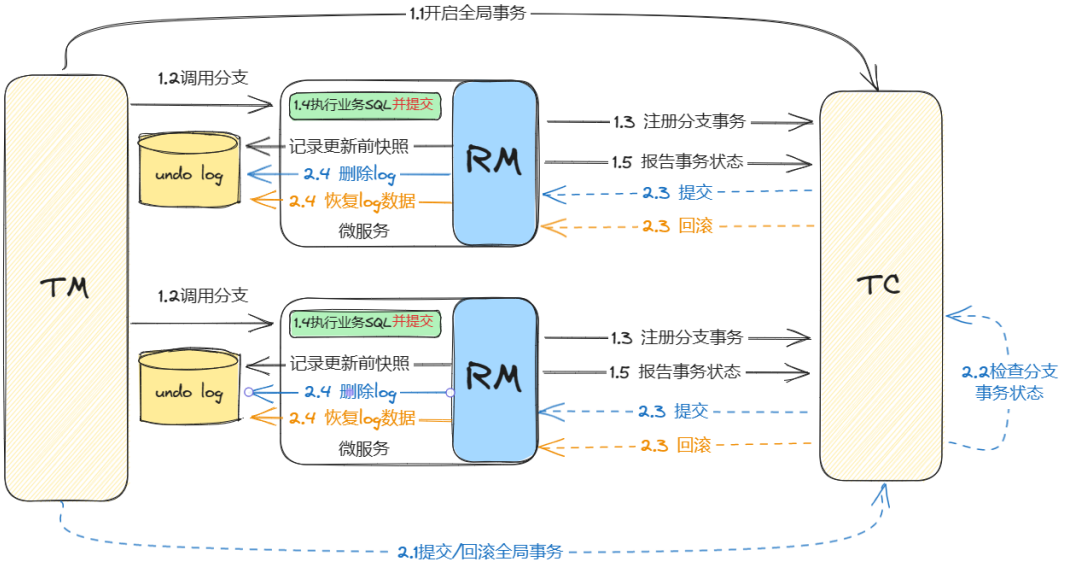
需要注意的是 seata 的 undo log 和 MySQL 的 undo log 虽然名字一样，但是结构却完全不同；

Seata 的 undo log 结构如下：

```
{
  "branchId":641789253,
  "undoItems":[
    {
      "afterImage":{
        "rows":[
          {…}
        ],
        "tableName":"product"
      },
      "beforeImage":{
        "rows":[
          {…}
        ],
        "tableName":"product"
      },
      "sqlType":"UPDATE"
    }
  ],
  "xid":"xid:xxx"
}
```


其中，XID 是分布式事务的 ID，branch_id 是分支事务的 ID（即本地事务的 ID），beforeImage 和 afterImage 为数据的前后镜像；

（3）AT 模式的流程



① 首先事务管理者会向事务协调者通知，并开启事务；

② 然后事务管理者会调用分布式事务的每一个数据操作

③ 每个数据操作在执行本地事务前，都会通知事务协调者本地事务已经开始，并直接提交本地事务并记录 undo log 日志，然后通知事务协调者本地事务执行的情况；

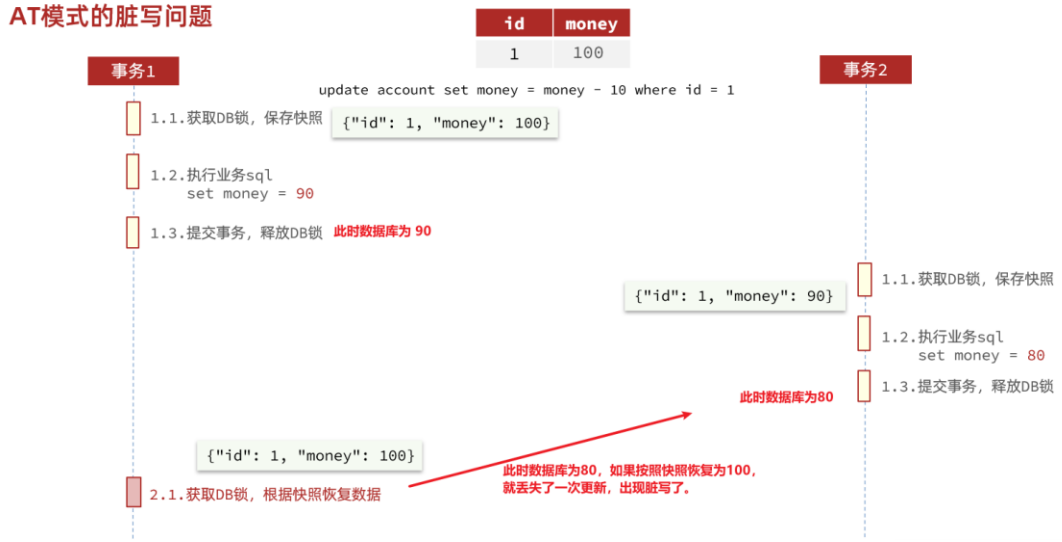
④ 在所有的数据操作都执行完毕后，事务管理者会向事务协调者发送分布式事务的提交/回滚操作通知；

⑤ 事务协调者会检测每个本地事务的返回结果：

- 如果全部成功，则提交分布式事务，删除每一个本地事务的 undo log
- 如果有一个失败，则回滚分布式事务，利用 undo log 回滚每一个本地事务，并且删除 undo log

（4）分布式事务，操作同一条数据的更新失败问题

由于 AT 模式强依赖于关系型数据库，已提交的数据和利用 undo log 回滚期间，数据是不加锁的，也就是说在本地事务提交后，分布式事务执行完毕之前，该数据是可以被其他程序读写的，就会导致更新失败（或者脏写）的问题。具体情况如下：



如上图：

- 分布式事务将 ID 为 1，数据为 100 更新为 90，并记录 seata 的 undo log 日志，但是分布式事务并没有结束；
- 然后其他事务 会将 ID 为 1，由 90 更新为 80；
- 分布式事务继续，然后回滚操作，将 ID 为 1 的数据，根据分布式事务 ID，查询 undo log 日志中的快照，回滚为 100，分布式事务完成；
- 这时，其他事务的更新失败，或者我们认为是分布式事务的脏写

（5）问题的解决

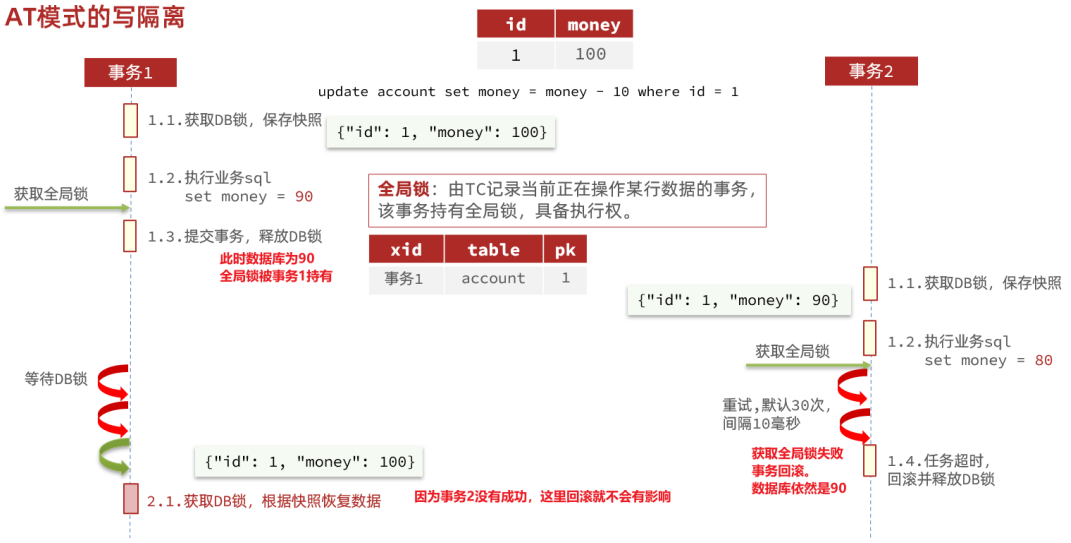
为了解决上述更新失败的问题，AT 模式引入了全局锁；

全局锁通常是由事务协调者维护的，由三个字段组成的关系型数据库表，三个字段分别为：

- 分布式事务 ID
- 该分布式事务正在操作表数据库表名
- 该分布式事务正在操作的数据库表中的某一行的主键

xid	table	pk
事务1	account	1

全局锁的使用流程如下：



① 假设有两个分布式事务，更新同一张表的同一条数据；

② 分布式事务 1 先获取到 DB 锁，行锁或表锁，锁定该行数据；

③ 然后执行 SQL，修改数据

④ 在本地事务提交之前，会通过事务协调者查询该条数据是否被其他分布式事务占用，没有则通过事务协调者修改或者插入全局锁表中一条数据，表示当前分布式事务正在锁定某个表中的某条数据

⑤ 然后本地事务进行提交，并记录 undo log 日志，释放 DB 锁；

⑥ 此时如果有其他分布式事务 2 来访问这条数据

⑦ 同样会首先获取到 DB 锁，即行锁或表锁，锁定该行数据；

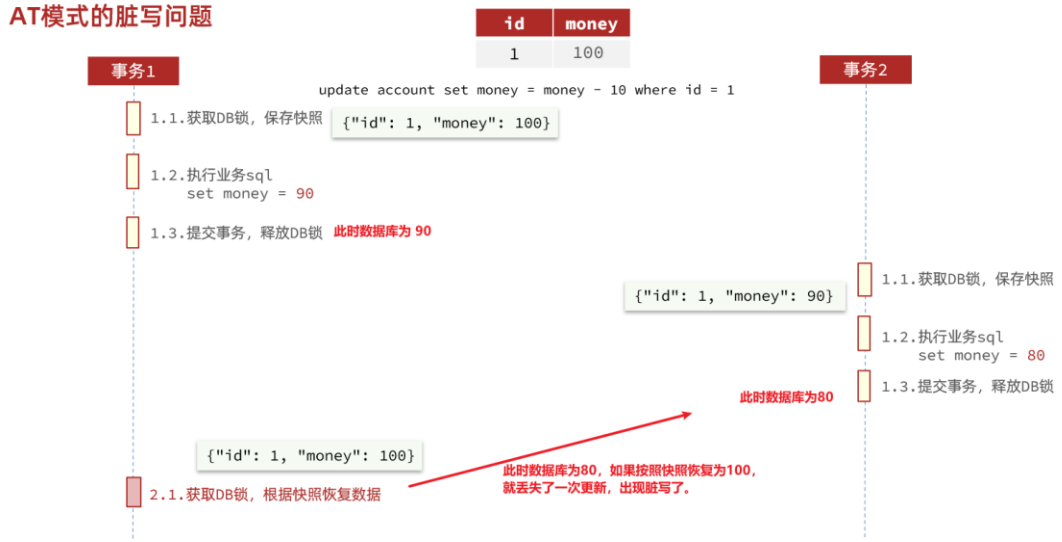
⑧ 然后执行分布式事务 2 的 SQL，修改数据；

⑨ 在本地事务提交之前，会通过事务协调者查询该条数据是否被其他分布式事务占用，查询到该数据被事务 1 所占用，无法获取全局锁，无法提交；线程将不断地去查询该数据的全局锁是否被释放，默认 30 次，超过 30 次全局锁还没被释放，本地事务就会回滚；

⑩ 分布式事务 1 的事务管理者向事务协调者发送提交/回滚命令，假设分布式事务 1 发生回滚，本地事务会根据事务协调者维护的全局锁表中的信息，到 seata 的 UNDO log 中查询到该条数据的前镜像，进行回滚；并删除掉全局锁中的相关记录，释放全局锁；

（6）带来的新的问题

① 分布式事务的全局锁是 seata 来维护的，所以非分布式事务和分布式事务修改同一数据时，容易出现更新失败的情况



1）假设有分布式事务 1 和非分布式事务 2，更新同一条数据 100；

2）分布式事务 1 的本地事务在将数据修改为 90 后，通过事务协调者添加分布式锁；

3）分布式事务 1 的本地事务提交事务，并记录 undo log，但分布式事务此时并没有进行提交或回滚；

4）此时其他非分布式事务 2 将该数据由 90 更新为 80.并且提交；

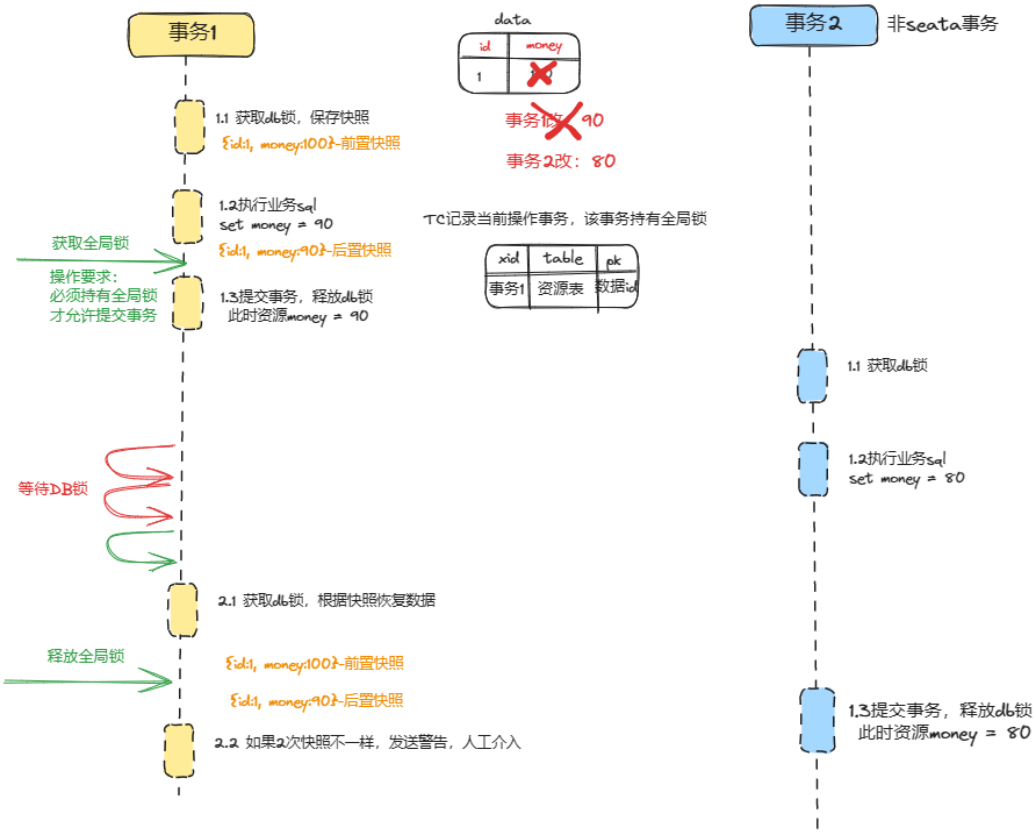
5）然后分布式事务 1 进行回滚，本地事务根据分布式事务 ID，到 undo log 表中查询到该数据的前镜像 100，进行回滚；分布式事务 1 执行完毕；

6）非分布式事务 2 更新失败；

② 解决上述问题的办法就是在分布式事务的本地事务回滚前，用 undo log 的后镜像和现在的数据进行比较；

如果相同，就进行回滚；

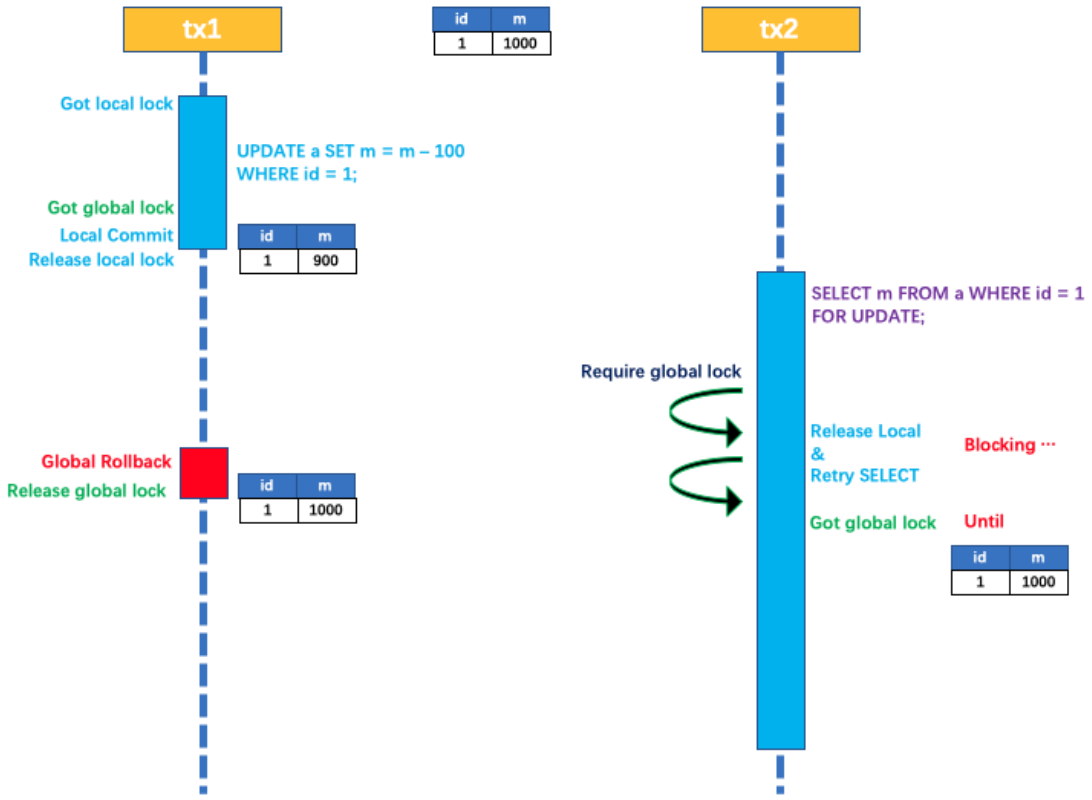
如果不相同，就抛异常或通知人工处理；



（7）写隔离/读隔离

上面介绍了全局锁的引入，解决了写隔离的问题，即同一个数据不会被脏写；下面介绍读隔离的问题；

SELECT FOR UPDATE 语句的执行会申请全局锁 ，如果全局锁被其他分布式事务持有，则释放 DB 锁并重试，直到获取到全局锁并查询成功；



（8）优缺点

① 优点

- 一阶段完成直接提交事务，释放数据库资源，性能比较好
- 利用全局锁实现读写隔离
- 没有代码侵入，框架自动完成回滚和提交

② 缺点

- 两阶段之间属于软状态，属于最终一致
- 框架的快照功能会影响性能，但比 XA 模式要好很多

（9）实现方式

实现方式很简单，只需要修改 YML 的分布式事务模式，并且创建框架指定的 undo log 表和全局锁表，使用@GlobalTransactional 来开启分布式事务即可

① 修改 YML 文件

```
seata:
  data-source-proxy-mode: AT # 默认就是AT
```

② 创建 undo log 表和全局锁表

```sql
DROP TABLE IF EXISTS `undo_log`;
CREATE TABLE `undo_log`  (
  `branch_id` bigint(20) NOT NULL COMMENT 'branch transaction id',
  `xid` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT 'global transa
  `context` varchar(128) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT 'undo_log
  `rollback_info` longblob NOT NULL COMMENT 'rollback info',
  `log_status` int(11) NOT NULL COMMENT '0:normal status,1:defense status',
  `log_created` datetime(6) NOT NULL COMMENT 'create datetime',
  `log_modified` datetime(6) NOT NULL COMMENT 'modify datetime',
  UNIQUE INDEX `ux_undo_log`(`xid`, `branch_id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT = 'AT transaction mode

-- ----------------------------
-- Records of undo_log
-- ----------------------------
```

undo log

全局锁

```sql
-- ----------------------------
-- Table structure for lock_table
-- ----------------------------
DROP TABLE IF EXISTS `lock_table`;
CREATE TABLE `lock_table`  (
  `row_key` varchar(128) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  `xid` varchar(96) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `transaction_id` bigint(20) NULL DEFAULT NULL,
  `branch_id` bigint(20) NOT NULL,
  `resource_id` varchar(256) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `table_name` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `pk` varchar(36) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `gmt_create` datetime NULL DEFAULT NULL,
  `gmt_modified` datetime NULL DEFAULT NULL,
  PRIMARY KEY (`row_key`) USING BTREE,
  INDEX `idx_branch_id`(`branch_id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Compact;
```

③ 使用@GlobalTransactional 来开启分布式事务

```java
@GlobalTransactional
public void purchase(String userId, String commodityCode, int orderCount, boolean rollback) {
    String result = stockFeignClient.deduct(commodityCode, orderCount);

    if (!"SUCCESS".equals(result)) {
        throw new RuntimeException("库存服务调用失败,事务回滚!");
    }
    result = orderFeignClient.create(userId, commodityCode, orderCount);
    if (!"SUCCESS".equals(result)) {
        throw new RuntimeException("订单服务调用失败,事务回滚!");
    }

    if (rollback) {
        throw new RuntimeException("Force rollback ... ");
    }
}
```

4．Seata TCC 模式

（1） 概述

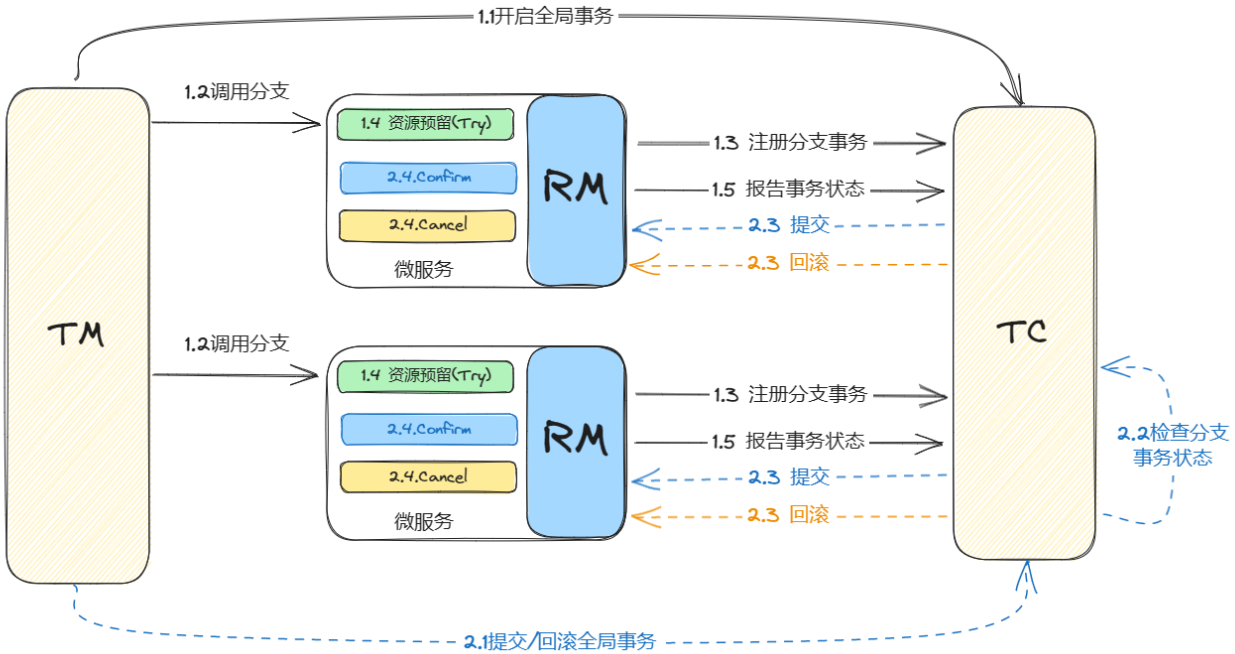
TCC 模式的总体流程和 AT 模式非常相似，只不过 AT 模式的细节是 seata 框架写好的，TCC 模式的细节则需要程序员自己来写；

TCC 即 try-confirm-cancel,是对 TCC 模式下，本地事务两阶段操作的总结；

- 第一阶段：
  - try：检测资源是否充足并且预留资源

- 第二阶段：
  - Confirm：如果第一阶段全部成功则进行提交
  - Cancel：如果第一阶段存在失败，则进行回滚

（2）原理



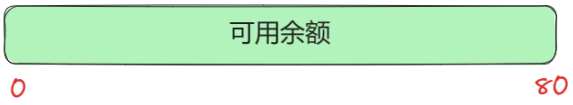
1) 首先事务管理者会向事务协调者通知，并开启事务；
2) 然后事务管理者会调用分布式事务的每一个数据操作；
3) 每个数据操作在执行本地事务前，都会通知事务协调者本地事务已经开始，并进行资源的检测及预留，然后通知事务协调者执行情况；
4) 在所有的数据操作都执行完毕后，事务管理者会向事务协调者发送分布式事务的提交/回滚操作通知；
5) 如果所有的检测和预留资源都成功了，那么就进行提交，清除预留资源；如果存在检测或预留资源失败的情况，那么就进行回滚，释放预留资源；

（3）TCC 模式 try-confirm-cancel 详解

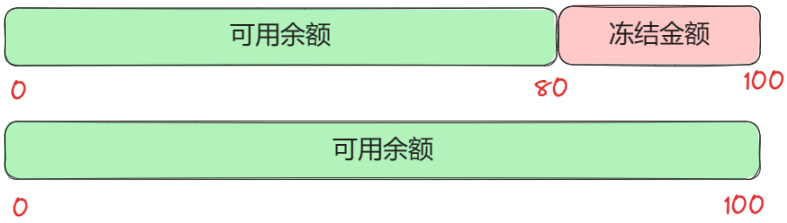
需求：RM：account-service 服务账户余额在TCC模式中的变化， 余额：100，扣款20

Try：检查余额是否充足，扣款20<余额100，满足条件

Confirm：直接扣减20，余额剩下80



cancel：回滚，解冻冻结金额，余额恢复100



比如本地事务要对 ID 为 1 的数据进行扣款 20 操作：

① 首先会检查余额是否充足，如果余额充足，就对该条数据冻结金额增加 20，可用余额减少 20，关系型数据库就进行提交，并将操作结果返回给事务协调者；

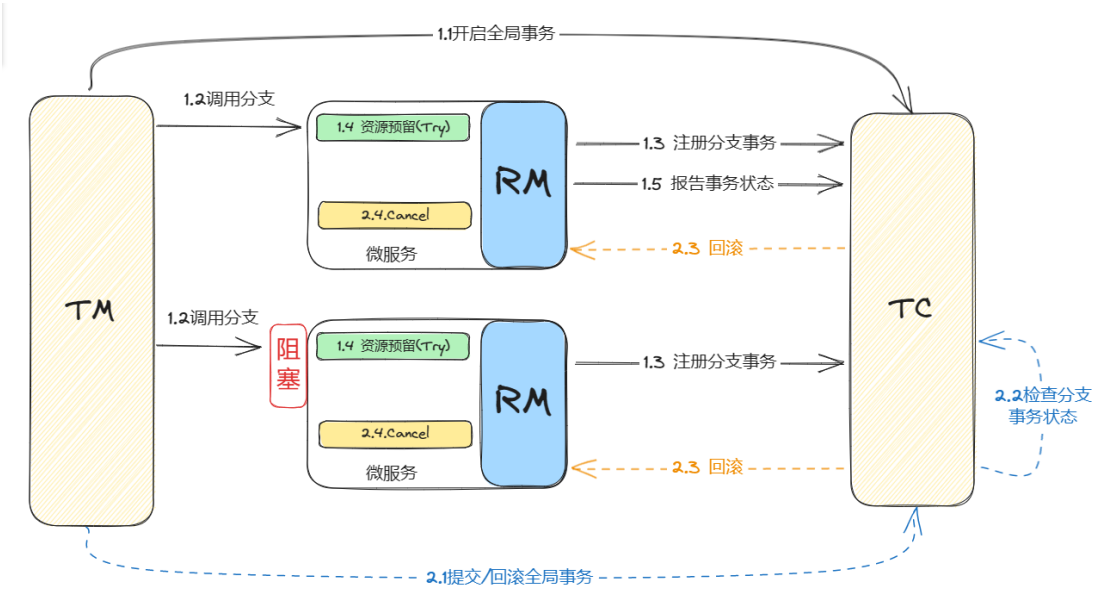
② 然后事务协调者在接收到事务管理者的二阶段通知，即进行分布式事务的提交或回滚，事务协调者就会根据各个本地事务反馈的信息来决定是提交还是回滚；

③ 如果要进行提交，则冻结金额直接减 20，可用余额不变

④ 如果要进行回滚，则冻结金额减 20，可用余额加 20

（4） TCC 模式的常见问题

① 空回滚

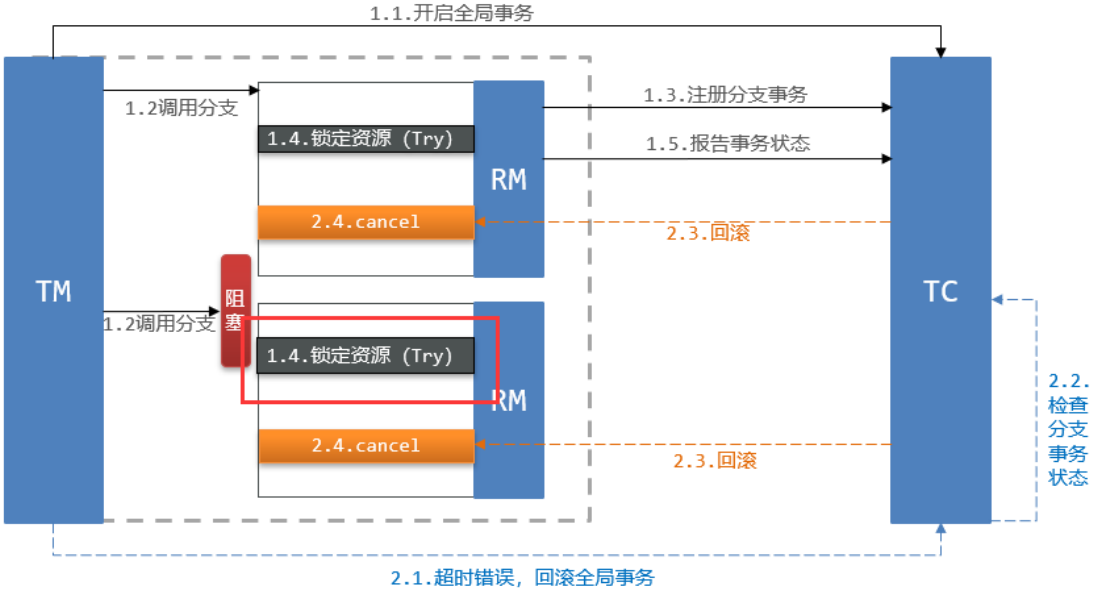


在 TCC 模式的第一阶段，如果由于网络延迟的问题，事务管理者调用本地事务出现了延迟、阻塞的情况，这样本地事务并没有进行资源的冻结；

在 TCC 模式的第二阶段，出现阻塞的本地事务接收到了事务协调者回滚的命令，就会出现空回滚情况；

② 业务悬挂

业务悬挂是伴随着空回滚出现的，当空回滚完成后，之前阻塞的事务管理者和本地事务调用链路才请求通，本地事务会执行 try 逻辑进行资源的预留和冻结，但是该分布式事务已经执行完毕，冻结的资源无法释放，造成业务悬挂；i



③ 重试幂等

在 TCC 模式下，本地事务的二阶段（confirm/cancel）阶段是一定要成功的，因为第一阶段（try）阶段预留冻结的资源一定要释放

所以事务协调者引入了重试机制，当发送的（confirm/cancel）命令未及时响应后，就会继续再次发送，这样的话当出现网络延迟时，本地事务就会接收到重复的 confirm/cancel 命令，造成问题；

（5）问题的解决

为了解决上述问题，在本地事务引入事务状态表，通常包含事务 ID，冻结金额，所处状态及其他辅助信息

1）try: 记录该事务 ID，预留的金额，并且记录状态为 TRY；扣减可用金额；

2）confirm：根据事务 ID 删除表中的冻结预留记录；

3）cancel：将事务 ID 的状态记录为 cancel，冻结金额置为 0，并且恢复可用金额；

4）解决空回滚问题，只需要在 cancel 阶段，根据事务 ID 查询该记录是否存在，如果不存在，说明并没有执行 try，需要空回滚；

5）解决业务悬挂问题，只需要在 try 阶段，根据事务 ID 查询是否存在该记录，如果存在并且状态为 cancel，则拒绝执行 try 操作；

6）解决重试幂等问题，只需要借助 redis，记录一下执行过的状态值即可

（6）TCC 模式的具体实现

① TCC 模式是在 AT 模式的基础上改造而来的，所以 YML 文件的分布式模式仍然为 AT 模式

```yaml
seata:
  data-source-proxy-mode: AT # 默认就是AT
```

② 然后就需要在每个本地事务手动实现 try、confirm、cancel 接口，通常使用@LocalTCC 来修饰接口

```java
*/
@LocalTCC
public interface IStockTCCService  {
    /**
     * try-预扣款
     */
    @TwoPhaseBusinessAction(name="tryDeduct", commitMethod = "confirm", rollbackMethod = "cancel")
    void tryDeduct(@BusinessActionContextParameter(paramName = "commodityCode") String commodityCode,
                   @BusinessActionContextParameter(paramName = "count") int count);

    /**
     * confirm-提交
     * @param ctx
     * @return
     */
    boolean confirm(BusinessActionContext ctx);

    /**
     * cancel-回滚
     * @param ctx
     * @return
     */
    boolean cancel(BusinessActionContext ctx);
}
```

③ 如果要预防空回滚，业务悬挂等，还需要在每个本地事务创建事务状态控制表，并编写相应的业务逻辑

```sql
CREATE TABLE `t_account_tx` (
  `id` bigint NOT NULL AUTO_INCREMENT COMMENT '主键',
  `tx_id` varchar(100) NOT NULL  COMMENT '事务id',
  `freeze_money` int DEFAULT NULL COMMENT '冻结金额',
  `state` int DEFAULT NULL COMMENT '状态 0try 1confirm 2cancel',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

④ 同样，分布式事务的开启只需要使用@GlobalTransactional 来开启即可

```java
@GlobalTransactional
public void purchase(String userId, String commodityCode, int orderCount, boolean rollback) {
    String result = stockFeignClient.deduct(commodityCode, orderCount);

    if (!"SUCCESS".equals(result)) {
        throw new RuntimeException("库存服务调用失败,事务回滚!");
    }
    result = orderFeignClient.create(userId, commodityCode, orderCount);
    if (!"SUCCESS".equals(result)) {
        throw new RuntimeException("订单服务调用失败,事务回滚!");
    }

    if (rollback) {
        throw new RuntimeException("Force rollback ... ");
    }
}
```

（7）优缺点

- 优点
  - 一阶段就完成了数据的提交，并没有锁定资源，且没有生成快照等其他文件，性能最好；
  - 并不依赖于关系型数据库，扩展强
- 缺点
  - 事务属于最终一致，且代码量比较大
  - Confirm、cancel 阶段需要做好幂等处理

5．Seata saga 模式

（1） 简介

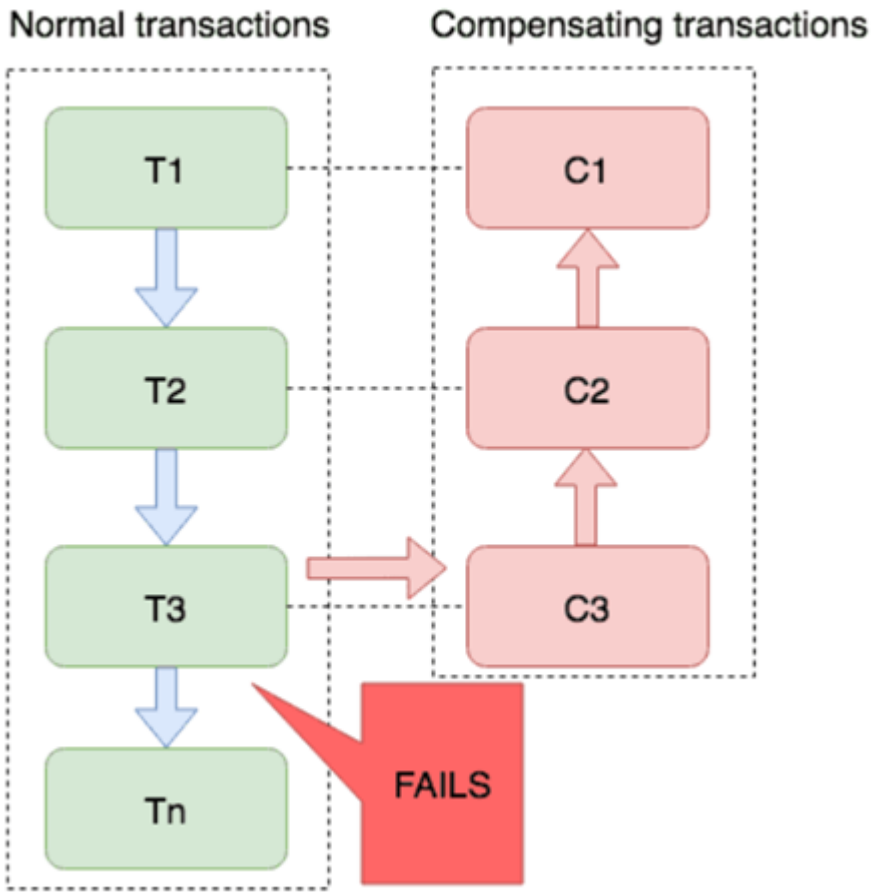
Sage 模式是适用于长事务模式，应用场景很少，常见于涉及多个 API 调用的情况，如多个银行间互相转账，MQ 等。

（2） 工作流程

Sage 同样属于二阶段提交的补偿型事务，相对于 TCC 来说，saga 要更简单一点

① 第一阶段：saga 直接完成数据的操作，并且提交事务

② 第二阶段：如果每个阶段都成功了就什么都不做，如果执行链路有一个失败了，则逆向链路执行补偿操作



（3）优缺点

①优点

- 无锁，吞吐量高，实现简单

②缺点

- 无锁，数据不隔离，容易发生脏写