

ch06-分布式事务

公众号：锋哥聊编程

主讲：小锋



1、本地事务与分布式事务

本地事务

本地事务，也就是传统的**单机事务**。在传统数据库事务中，必须要满足四个原则：

事务四大特性

原子性(Atomicity)

事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性 (Consistency)

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

隔离性 (Isolation)

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

持久性 (Durability)

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

分布式事务

分布式事务是指事务的**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。

随着微服务架构的普及，大型业务域往往包含很多服务，一个业务流程需要由多个服务共同参与完成。在特定的业务场景中，需要保障多个服务之间的数据一致性。

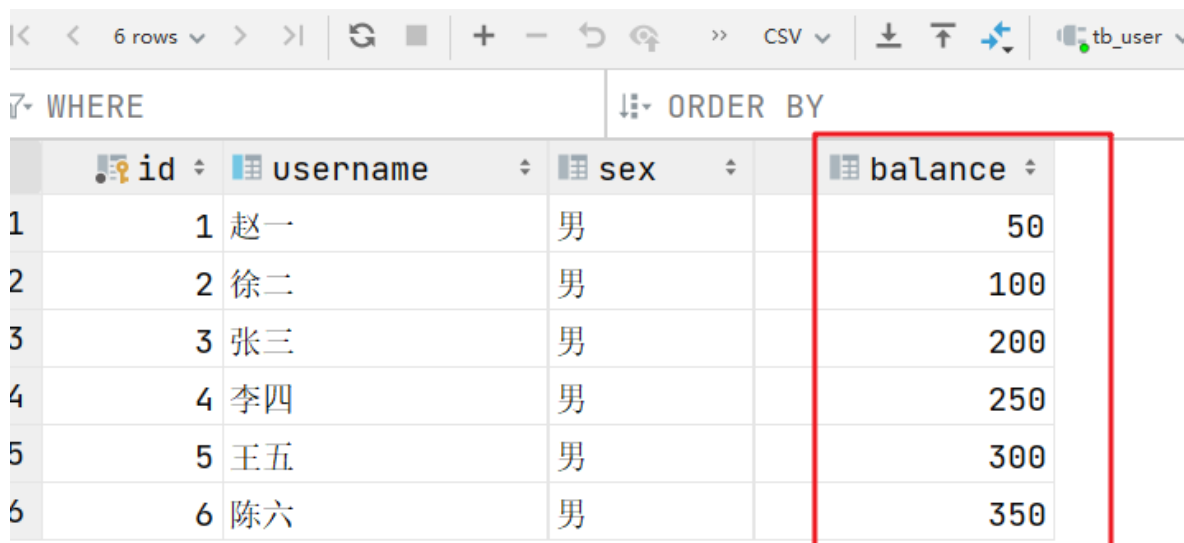
例如：在大型电商系统中，下单接口通常会扣减库存、减去优惠、生成订单 id,, 而订单服务与库存、优惠、订单 id 都是不同的服务，下单接口的成功与否，不仅取决于本地的 db 操作，而且依赖第三方系统的结果，这时候分布式事务就保证这些操作要么全部成功，要么全部失败。

所以本质上来说，**分布式事务就是为了保证不同数据库的数据一致性**。

2、编写分布式事务演示代码

修改表

在mafeng-user添加balance余额字段



	id	username	sex	balance
1	1	赵一	男	50
2	2	徐二	男	100
3	3	张三	男	200
4	4	李四	男	250
5	5	王五	男	300
6	6	陈六	男	350

user服务添加方法

在mafeng-user服务添加扣减用户余额方法

UserController:

```

/**
 * 扣减账户余额
 */
@PostMapping("/deduct/{id}/{money}")
public void deduct(@PathVariable("id") Long id,@PathVariable("money") Long
money){
    userService.deduct(id,money);
}

```

UserService:

```

@Override
public void deduct(Long id, Long money) {
    User user = getById(id);
    user.setBalance(user.getBalance()-money);
    updateById(user);
}

```

feign接口添加方法

mafeng-order服务的UserClient接口添加方法:

```

/**
 * 扣减账户余额
 */
@PostMapping("/deduct/{id}/{money}")
public void deduct(@PathVariable("id") Long id,@PathVariable("money") Long
money);

```

order服务添加方法

mafeng-order服务添加下订单方法

OrderController:

```

/**
 * 下订单
 * @param order
 * @return
 */
@PostMapping("/save")
public String save(@RequestBody Order order){
    try {
        ordersService.saveOrder(order);
        return "下单成功";
    } catch (Exception e) {
        return "下单失败";
    }
}

```

OrderService:

```

@Override
@Transactional
public void saveOrder(Order order) {
    //保存订单
    save(order);

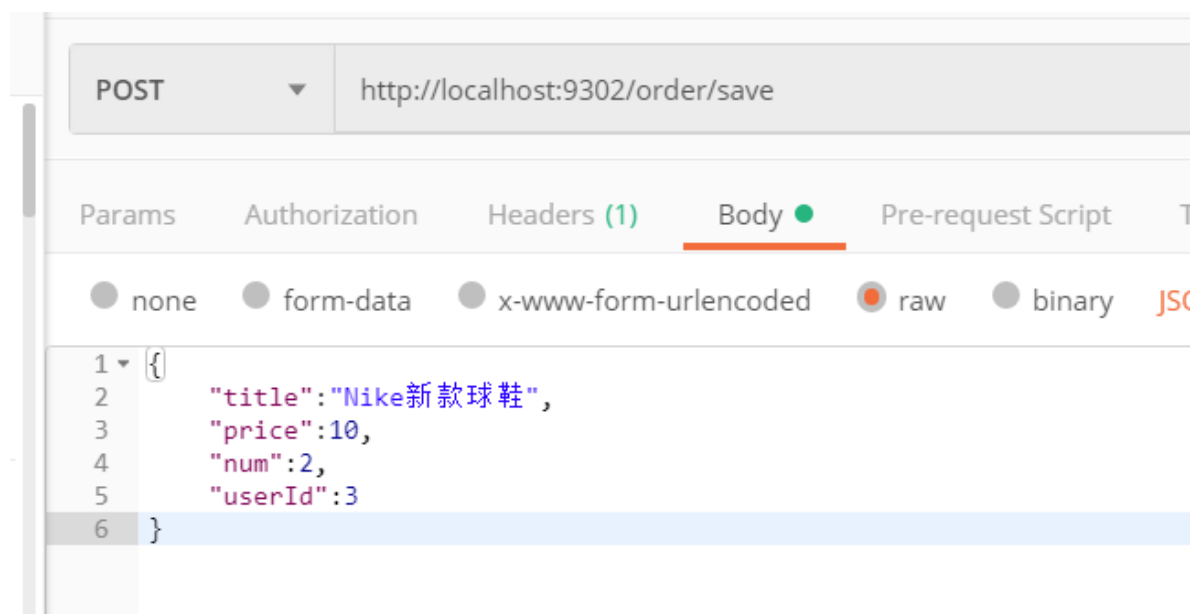
    //扣减用户余额
    userClient.deduct(order.getUserId(), order.getPrice() * order.getNum());

    int i = 100/0; //异常发生
}

```

注意：该方法在最后模拟异常发生，且当前业务方法已经添加Transactional注解

下单接口测试



测试结果发现，mafeng-order表数据可以被回滚，但mafeng-user表记录不能被回滚！两张表数据不一致！

3、CAP理论与分布事务解决方案

回顾CAP理论

CAP理论是埃里克·布鲁尔（Eric Brewer）提出的理论。在分布式系统领域，特别是大数据应用、NOSQL数据库等分布式复杂系统的设计方面给出了很好的指导方针。CAP理论的核心思想就是：**强数据一致性、高可用、分区容忍性之间，只能三选二。也就是说：当你选择其中的两项作为来满足你的系统需求，就必须舍弃第三项，三项无法同时满足。**

强数据一致性(High Consistency)

这个C必须是**强数据一致性**。如果应用仅仅满足了弱一致性或者最终一致性，它并不满足CAP的C特性。

- **强一致性**：强一致性是一种一致性模型，在该模型中，对分布式系统的所有后续访问将在更新后始终返回更新后的值。不能出现脏读或者幻读的现象，这种强一致性通常是通过数据锁(悲观锁)来实现的。

- **弱一致性：** 这是一种用于分布式计算的一致性模型，其中后续访问可能并不总是返回更新后的值。可能在某一时间段会有数据不一致的响应。这是一种尽力而为的一致性模型，分布式应用之间彼此通过网络交换数据，但是因为没有数据锁定机制，可能导致不同节点同一时间对外提供的服务数据不一致。
- **最终一致性：** 最终一致性是弱一致性中的一种特殊类型。这种模型的数据一致性，通常是通过消息队列来实现的，数据提供方将数据放入消息队列，他就不再去关注该数据是否被处理。消息队列保证数据被数据消费方成功处理一次（并且值处理一次）。至于该数据什么时候被成功处理，那就不一定了。

高可用性 (High Availability)

高可用通俗的说，就是任何时间都可用，任何请求都可以得到响应。怎么保障高可用？应用水平扩展n个节点，其中1个或者几个节点挂了，其他的节点仍然可以支撑整体服务的运行。

什么样的系统是CA系统？ 比如：我们的关系型数据库Oracle、MySQL，都是可以搭建高可用环境的，也就是满足A特性。

分区容忍性 (Partition Tolerance)

- AP系统：在分布式系统中某些节点**网络不可达或故障**的时候，仍然可以对外**正常提供服务**的系统。。
- CP系统：在分布式系统中某些节点**网络不可达或故障**的时候，仍然以**保证系统节点之间的数据一致性**优先，可能导致整个服务短暂锁定或者宕机。

那么，什么是分区容忍性（P）？通常是指分布式系统部分节点由于网络故障导致不可达，仍然可以对外提供满足一致性或者高可用的服务。

分布式事务解决思路

借鉴CAP定理，对于分布式事务，我们有两种解决思路：

- AP模式：各子事务分别执行和提交，允许出现结果不一致，然后采用弥补措施恢复数据即可，实现最终一致。（强可用，弱一致）
- CP模式：各个子事务执行后互相等待，同时提交，同时回滚，达成强一致。但事务等待过程中，处于弱可用状态。（强一致，弱可用）

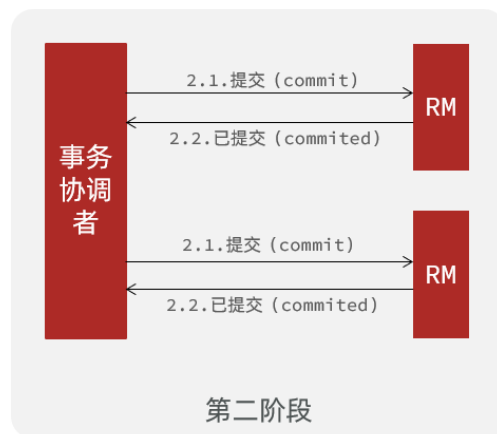
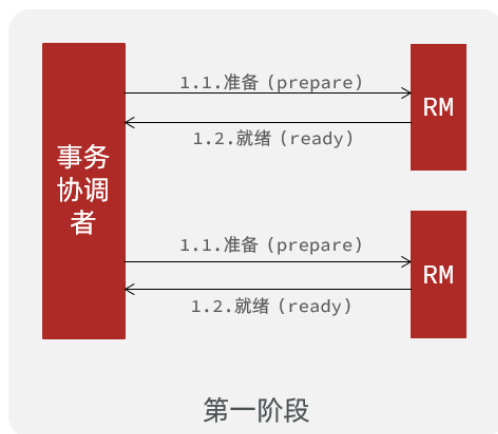
分布式事务解决方案

XA两阶段提交 (2PC)

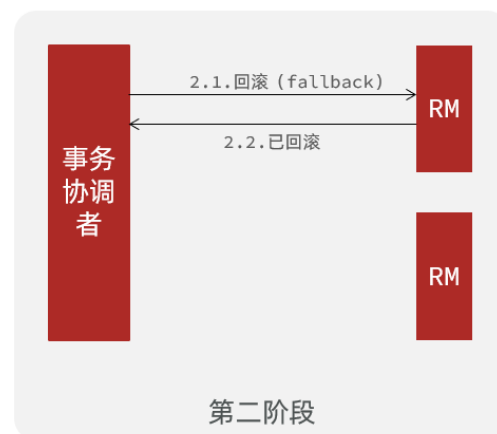
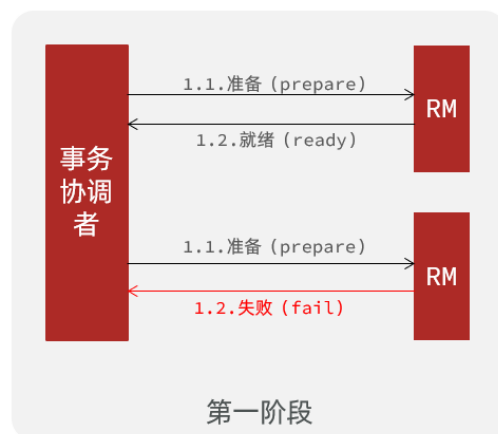
XA 规范 是 X/Open 组织定义的分布式事务处理（DTP，Distributed Transaction Processing）标准，XA 规范 描述了全局的TM与局部的RM之间的接口，几乎所有主流的关系型数据库都对 XA 规范 提供了支持。

XA是规范，目前主流数据库都实现了这种规范，实现的原理都是基于两阶段提交。

正常情况：



异常情况:



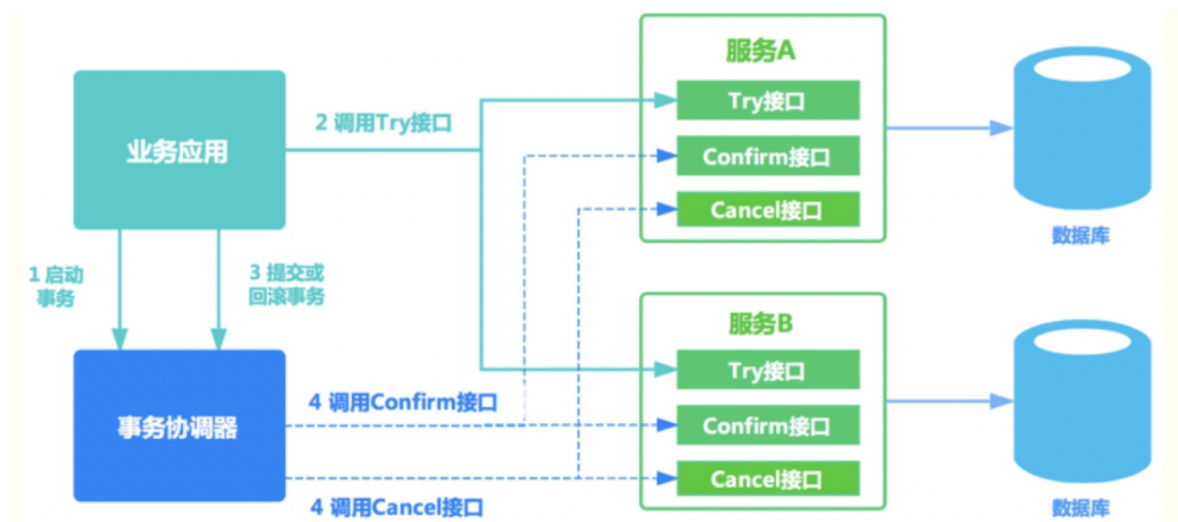
XA两阶段提交的优点:

- 事务的强一致性，满足ACID原则。
- 常用关系型数据库都支持，实现简单

XA两阶段提交的缺点:

- 因为一阶段需要锁定数据库资源，等待二阶段结束才释放，性能较差
- 依赖关系型数据库实现事务

补偿事务 (TCC)



TCC (Try Confirm Cancel) 方案是一种应用层面侵入业务的两阶段提交。TCC采用了补偿机制，其核心思想就是针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为Try、Confirm和Cancel三个阶段。

- 1) Try 阶段：主要是对业务系统做检测及资源预留；
- 2) Confirm 阶段：主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认Confirm阶段是不会出错的，即：只要Try成功，Confirm一定成功；
- 3) Cancel 阶段：主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

转账例子：假如 Bob 要向 Smith 转账，思路大概是：我们有一个本地方法，里面依次调用。

1. 首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来；
2. 在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。
3. 如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

TCC模式优点：

- 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

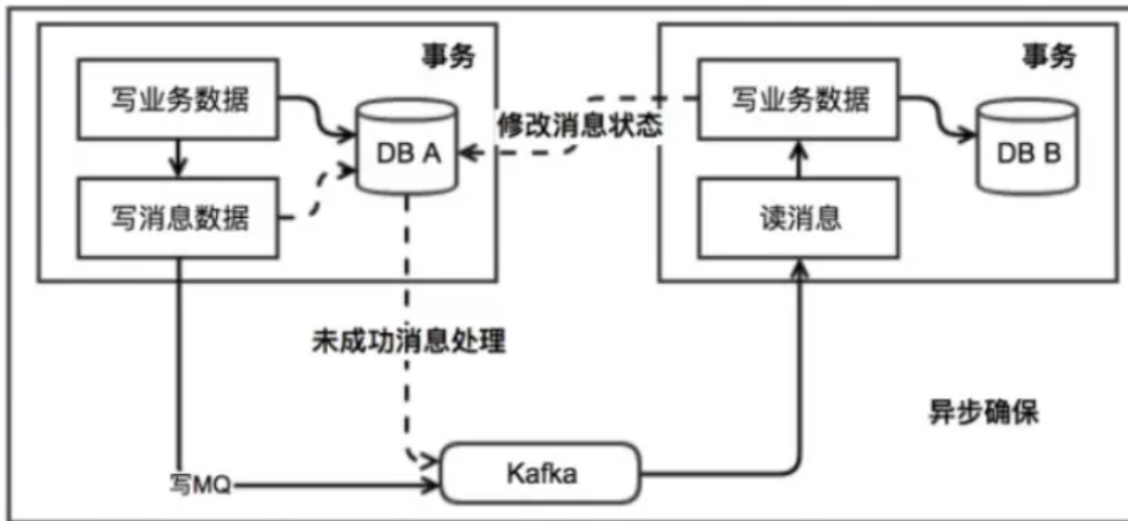
TCC模式缺点：

- 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

本地消息表（最终一致性）

本地消息表与业务数据表处于同一个数据库中，这样就能利用本地事务来保证在对这两个表的操作满足事务特性，并且使用了消息队列来保证最终一致性。

1. 在分布式事务操作的一方完成写业务数据的操作之后向本地消息表发送一个消息，本地事务能保证这个消息一定会被写入本地消息表中。
2. 之后将本地消息表中的消息转发到 Kafka 等消息队列中，如果转发成功则将消息从本地消息表中删除，否则继续重新转发。
3. 在分布式事务操作的另一方从消息队列中读取一个消息，并执行消息中的操作。



本地消息表的优点：

- 一种非常经典的实现，避免了分布式事务，实现了最终一致性。

本地消息表的缺点：

- 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

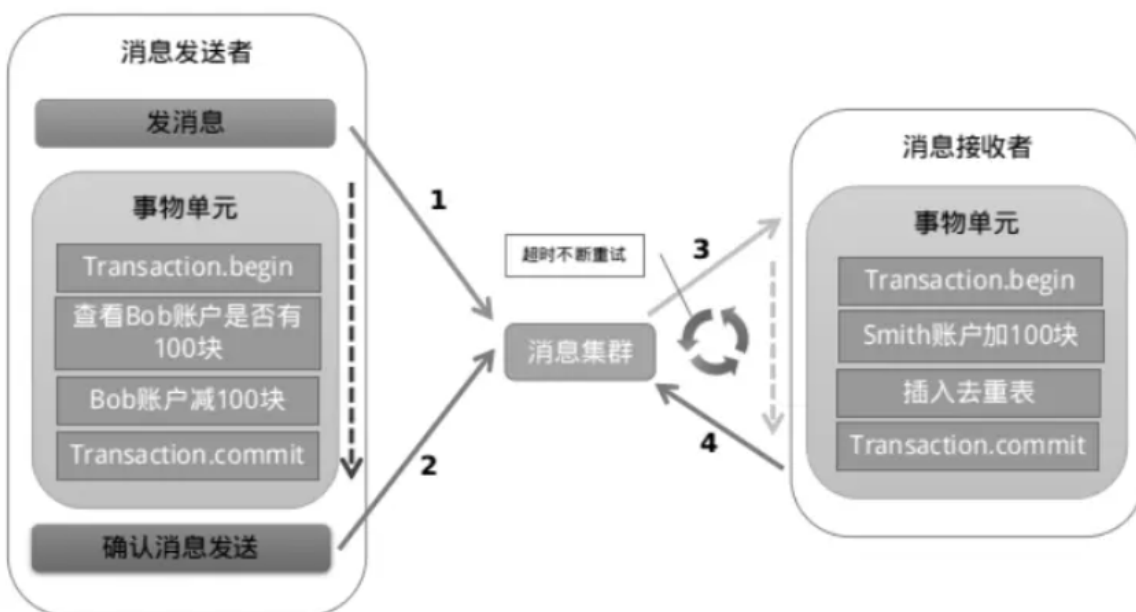
MQ 事务消息

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。



MQ 事务消息优点：

- 实现了最终一致性，不需要依赖本地数据库事务。

MQ 事务消息缺点：

- 实现难度大，主流MQ不支持，RocketMQ事务消息部分代码也未开源。

Seata框架

Seata是 2019 年 1 月份蚂蚁金服和阿里巴巴共同开源的分布式事务解决方案。致力于提供高性能和简单易用的分布式事务服务，为用户打造一站式的分布式解决方案。

官网地址：<http://seata.io/>。



Seata提供了四种不同的分布式事务解决方案：

- XA模式：强一致性分阶段事务模式，牺牲了一定的可用性，无业务侵入
- TCC模式：最终一致的分阶段事务模式，有业务侵入
- AT模式：最终一致的分阶段事务模式，无业务侵入，也是**Seata的默认模式**
- SAGA模式：长事务模式，有业务侵入

Seata优点：

- 应用层基于SQL解析实现了自动补偿，从而最大程度的降低业务侵入性；
- 将分布式事务中TC（事务协调者）独立部署，负责事务的注册、回滚；
- 通过全局锁实现了写隔离与读隔离。

Seata缺点：

- 性能损耗：全局锁的引入实现了隔离性，但带来的问题就是阻塞，降低并发性，尤其是热点数据，这个问题会更加严重。
- 回滚锁释放时间长：Seata在回滚时，需要先删除各节点的undo log，然后才能释放TC内存中的锁，所以如果第二阶段是回滚，释放锁的时间会更长。

- 死锁问题： Seata的引入全局锁会额外增加死锁的风险，但如果出现死锁，会不断进行重试，最后靠等待全局锁超时，这种方式并不优雅，也延长了对数据库锁的占有时间。

4、Seata架构

Seata事务管理中有三个重要的角色：

- **TC (Transaction Coordinator) - 事务协调者**

维护全局和分支事务的状态，驱动全局事务提交或回滚。

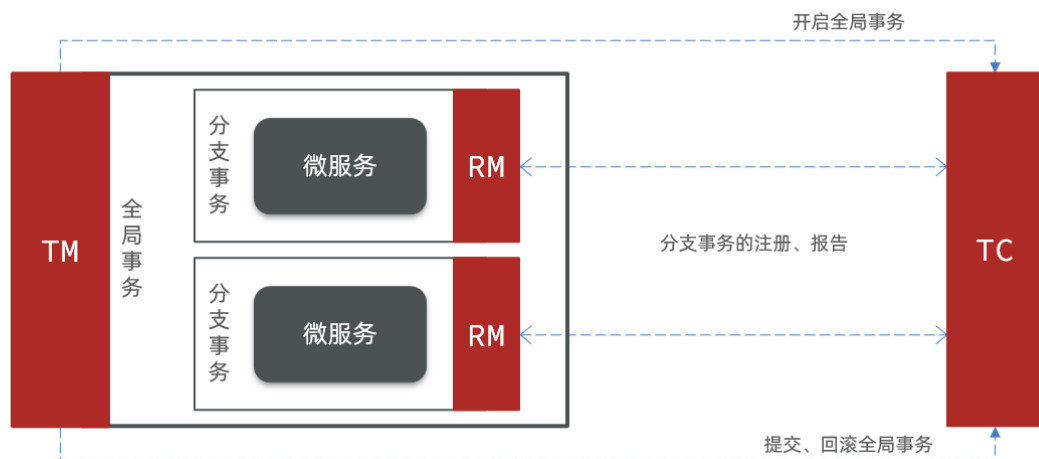
- **TM (Transaction Manager) - 事务管理器**

定义全局事务的范围：开始全局事务、提交或回滚全局事务。

- **RM (Resource Manager) - 资源管理器**

管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。

整体的架构如图：



5、搭建Seata的服务端

下载seata服务端镜像

```
docker pull seataio/seata-server:1.5.1
```

修改application.yml配置

```
server:
  port: 7091

spring:
  application:
    name: seata-server
```

```
logging:
  config: classpath:logback-spring.xml
  file:
    path: ${user.home}/logs/seata
  extend:
    logstash-appender:
      destination: 127.0.0.1:4560
    kafka-appender:
      bootstrap-servers: 127.0.0.1:9092
      topic: logback_to_logstash

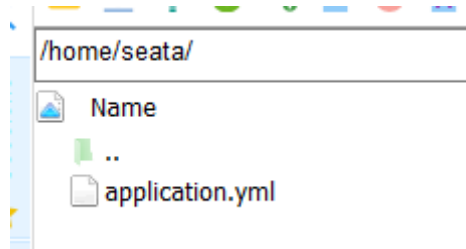
console:
  user:
    username: seata #seata-Web管理系统账号密码
    password: seata

seata:
  config:
    type: nacos
    nacos:
      server-addr: http://192.168.66.133:8861 #填线上Nacos的IP:PORT
      namespace: #命名空间，使用默认的不用填
      group: SEATA_GROUP #组
      username: nacos
      password: nacos
      data-id: seataServer.properties #seataServer配置文件
  registry:
    type: nacos
    nacos:
      application: seata-server #注册服务名称
      server-addr: http://192.168.66.133:8861 #填线上Nacos的IP:PORT
      group: SEATA_GROUP #组
      namespace: #命名空间，使用默认的不用填
      cluster: default
      username: nacos
      password: nacos
  store:
    mode: db
    db:
      datasource: druid
      db-type: mysql
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://192.168.66.133:3306/seata?serverTimezone=Asia/Shanghai #
      #填线上MySQL的IP:PORT，seata数据库名，要和前面插入seata表同一个名
      user: root #数据库账号
      password: root #数据库密码
      min-conn: 5
      max-conn: 100
      global-table: global_table
      branch-table: branch_table
      lock-table: lock_table
      distributed-lock-table: distributed_lock
      query-limit: 100
      max-wait: 5000

  security:
    secretKey: SeataSecretKey0c382ef121d778043159209298fd40bf3850a017
    tokenValidityInMilliseconds: 1800000
```

```
ignore:
  urls:
/,/**/*.*css,/**/*.*js,/**/*.*html,/**/*.*map,/**/*.*svg,/**/*.*png,/**/*.*ico,/console
-fe/public/**,/api/v1/auth/login
```

同时把application.yml文件上传到/home/seata/目录下



Nacos创建配置

在Nacos上面创建seataServer.properties文件，Group为SEATA_GROUP

Data ID: seataServer.properties

Group: SEATA_GROUP

```
#For details about configuration items, see https://seata.io/zh-cn/docs/user/configurations.html
#Transport configuration, for client and server
transport.type=TCP
transport.server=NIO
transport.heartbeat=true
transport.enableTmClientBatchSendRequest=false
transport.enableRmClientBatchSendRequest=true
transport.enableTcServerBatchSendResponse=false
transport.rpcRmRequestTimeout=30000
transport.rpcTmRequestTimeout=30000
transport.rpcTcRequestTimeout=30000
transport.threadFactory.bossThreadPrefix=NettyBoss
transport.threadFactory.workerThreadPrefix=NettyServerNIOWorker
transport.threadFactory.serverExecutorThreadPrefix=NettyServerBizHandler
transport.threadFactory.shareBossWorker=false
transport.threadFactory.clientSelectorThreadPrefix=NettyClientSelector
transport.threadFactory.clientSelectorThreadSize=1
transport.threadFactory.clientWorkerThreadPrefix=NettyClientWorkerThread
transport.threadFactory.bossThreadSize=1
transport.threadFactory.workerThreadSize=default
transport.shutdown.wait=3
transport.serialization=seata
transport.compressor=none

#Transaction routing rules configuration, only for the client
service.vgroupMapping.default_tx_group=default
#If you use a registry, you can ignore it
service.default.grouplist=192.168.66.133:8091 #修改为线上IP:PORT
service.enableDegrade=false
service.disableGlobalTransaction=false

#Transaction rule configuration, only for the client
```

```
client.rm.asyncCommitBufferLimit=10000
client.rm.lock.retryInterval=10
client.rm.lock.retryTimes=30
client.rm.lock.retryPolicyBranchRollbackOnConflict=true
client.rm.reportRetryCount=5
client.rm.tableMetaCheckEnable=true
client.rm.tableMetaCheckerInterval=60000
client.rm.sqlParserType=druid
client.rm.reportSuccessEnable=false
client.rm.sagaBranchRegisterEnable=false
client.rm.sagaJsonParser=fastjson
client.rm.tccActionInterceptorOrder=-2147482648
client.tm.commitRetryCount=5
client.tm.rollbackRetryCount=5
client.tm.defaultGlobalTransactionTimeout=60000
client.tm.degradeCheck=false
client.tm.degradeCheckAllowTimes=10
client.tm.degradeCheckPeriod=2000
client.tm.interceptorOrder=-2147482648
client.undo.dataValidation=true
client.undo.logSerialization=jackson
client.undo.onlyCareUpdateColumns=true
server.undo.logSaveDays=7
server.undo.logDeletePeriod=86400000
client.undo.logTable=undo_log
client.undo.compress.enable=true
client.undo.compress.type=zip
client.undo.compress.threshold=64k
#For TCC transaction mode
tcc.fence.logTableName=tcc_fence_log
tcc.fence.cleanPeriod=1h

#Log rule configuration, for client and server
log.exceptionRate=100

#Transaction storage configuration, only for the server. The file, DB, and redis
configuration values are optional.
store.mode=file
store.lock.mode=file
store.session.mode=file
#Used for password encryption
store.publickey=

#If `store.mode,store.lock.mode,store.session.mode` are not equal to `file`, you
can remove the configuration block.
store.file.dir=file_store/data
store.file.maxBranchSessionSize=16384
store.file.maxGlobalSessionSize=512
store.file.fileWriteBufferCacheSize=16384
store.file.flushDiskMode=async
store.file.sessionReloadReadSize=100

#These configurations are required if the `store mode` is `db`. If
`store.mode,store.lock.mode,store.session.mode` are not equal to `db`, you can
remove the configuration block.
store.db.datasource=druid
store.db.dbType=mysql
store.db.driverClassName=com.mysql.cj.jdbc.Driver
```

```

store.db.url=jdbc:mysql://192.168.66.133:3306/seata?
serverTimezone=Asia/Shanghai&useUnicode=true&rewriteBatchedStatements=true #修改
为线上IP:PORT
store.db.user=root #修改
store.db.password=root #修改
store.db.minConn=5
store.db.maxConn=30
store.db.globalTable=global_table
store.db.branchTable=branch_table
store.db.distributedLockTable=distributed_lock
store.db.queryLimit=100
store.db.lockTable=lock_table
store.db.maxwait=5000

#These configurations are required if the `store mode` is `redis`. If
`store.mode,store.lock.mode,store.session.mode` are not equal to `redis`, you
can remove the configuration block.
store.redis.mode=single
store.redis.single.host=127.0.0.1
store.redis.single.port=6379
store.redis.sentinel.masterName=
store.redis.sentinel.sentinelHosts=
store.redis.maxConn=10
store.redis.minConn=1
store.redis.maxTotal=100
store.redis.database=0
store.redis.password=
store.redis.queryLimit=100

#Transaction rule configuration, only for the server
server.recovery.committingRetryPeriod=1000
server.recovery.asyncCommittingRetryPeriod=1000
server.recovery.rollbackingRetryPeriod=1000
server.recovery.timeoutRetryPeriod=1000
server.maxCommitRetryTimeout=-1
server.maxRollbackRetryTimeout=-1
server.rollbackRetryTimeoutUnlockEnable=false
server.distributedLockExpireTime=10000
server.xaerNotaRetryTimeout=60000
server.session.branchAsyncQueueSize=5000
server.session.enableBranchAsyncRemove=false
server.enableParallelRequestHandle=false

#Metrics configuration, only for the server
metrics.enabled=false
metrics.registryType=compact
metrics.exporterList=prometheus
metrics.exporterPrometheusPort=9898

```

在Nacos创建 service.vgroupMapping.default_tx_group , Group为 SEATA_GROUP

Data ID: service.vgroupMapping.default_tx_group

Group: SEATA_GROUP

配置内容: default

default

创建seata数据库

在mysql数据库创建名为seata的数据库，sql如下：

```
-- ----- The script used when storeMode is 'db' -----
-----
-- the table to store GlobalSession data
CREATE TABLE IF NOT EXISTS `global_table`
(
    `xid`                VARCHAR(128) NOT NULL,
    `transaction_id`     BIGINT,
    `status`             TINYINT      NOT NULL,
    `application_id`     VARCHAR(32),
    `transaction_service_group` VARCHAR(32),
    `transaction_name`   VARCHAR(128),
    `timeout`            INT,
    `begin_time`         BIGINT,
    `application_data`   VARCHAR(2000),
    `gmt_create`         DATETIME,
    `gmt_modified`       DATETIME,
    PRIMARY KEY (`xid`),
    KEY `idx_status_gmt_modified` (`status`, `gmt_modified`),
    KEY `idx_transaction_id` (`transaction_id`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4;

-- the table to store BranchSession data
CREATE TABLE IF NOT EXISTS `branch_table`
(
    `branch_id`          BIGINT      NOT NULL,
    `xid`                VARCHAR(128) NOT NULL,
    `transaction_id`     BIGINT,
    `resource_group_id`  VARCHAR(32),
    `resource_id`        VARCHAR(256),
    `branch_type`        VARCHAR(8),
    `status`             TINYINT,
    `client_id`          VARCHAR(64),
    `application_data`   VARCHAR(2000),
    `gmt_create`         DATETIME(6),
    `gmt_modified`       DATETIME(6),
    PRIMARY KEY (`branch_id`),
    KEY `idx_xid` (`xid`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4;

-- the table to store lock data
CREATE TABLE IF NOT EXISTS `lock_table`
(
    `row_key`            VARCHAR(128) NOT NULL,
    `xid`                VARCHAR(128),
    `transaction_id`     BIGINT,
    `branch_id`          BIGINT      NOT NULL,
```

```

    `resource_id`      VARCHAR(256),
    `table_name`       VARCHAR(32),
    `pk`               VARCHAR(36),
    `status`           TINYINT      NOT NULL DEFAULT '0' COMMENT '0:locked
,1:rollbacking',
    `gmt_create`       DATETIME,
    `gmt_modified`     DATETIME,
    PRIMARY KEY (`row_key`),
    KEY `idx_status` (`status`),
    KEY `idx_branch_id` (`branch_id`),
    KEY `idx_xid_and_branch_id` (`xid` , `branch_id`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4;

CREATE TABLE IF NOT EXISTS `distributed_lock`
(
    `lock_key`         CHAR(20) NOT NULL,
    `lock_value`       VARCHAR(20) NOT NULL,
    `expire`           BIGINT,
    primary key (`lock_key`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4;

INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('AsyncCommitting', ' ', 0);
INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('RetryCommitting', ' ', 0);
INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('RetryRollbacking', ' ', 0);
INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('TxTimeoutCheck', ' ', 0);

```

启动seata服务端容器

```

docker run --name seata-server \
    -p 8091:8091 \
    -p 7091:7091 \
    -e SEATA_IP=192.168.66.133 \
    -e SEATA_PORT=8091 \
    -v /home/seata/application.yml:/seata-server/resources/application.yml
\
    -d seataio/seata-server:1.5.1

```

查看运行情况

seata服务端启动成功后，查看nacos服务列表可以看到

服务列表 | public

服务名称 分组名称 隐藏空服务: ☒

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值
seata-server	SEATA_GROUP	1	1	1	false

访问: <http://192.168.66.133:7091/>

账户/密码: seata

← → ↻ ▲ 不安全 | 192.168.66.133:7091/#/TransactionInfo

Seata

TransactionInfo

GlobalLockInfo

TransactionInfo / list

TransactionInfo

CreateTime -

status Whether to include branch sessions ☐

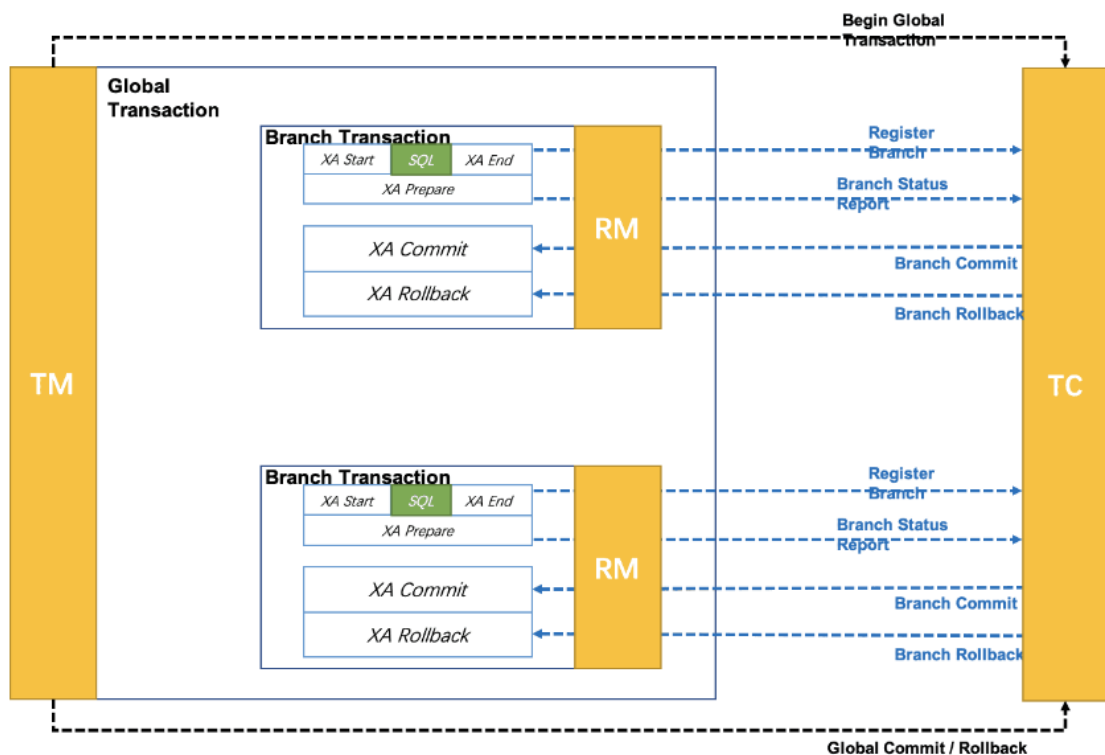
xid	transactionId	applicationId	transactionServiceGroup	transactionName	status	timeout	beginTime
No Data							

Items per page:

6、Seata分布式（一）XA模式

Seata的XA模式原理

在 Seata 定义的分布式事务框架内，利用事务资源（数据库、消息服务等）对 XA 协议的支持，以 XA 协议的机制来管理分支事务的一种 事务模式。



Seata的XA模式步骤

导入seata依赖

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
  <version>2021.0.1.0</version>
```

```

        <!--需要排除低版本的seata，手动引入1.5.1-->
        <exclusions>
            <exclusion>
                <artifactId>seata-spring-boot-starter</artifactId>
                <groupId>io.seata</groupId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>io.seata</groupId>
        <artifactId>seata-spring-boot-starter</artifactId>
        <version>1.5.1</version>
    </dependency>

```

配置seata

```

seata:
  enable: true
  application-id: seata-cilent-user
  tx-service-group: default_tx_group
  registry:
    type: nacos
    nacos:
      application: seata-server # seata 服务名
      # 非本地请修改具体的地址
      server-addr: http://192.168.66.133:8861
      group: SEATA_GROUP
  config:
    type: nacos
    nacos:
      # nacos ip地址
      server-addr: http://192.168.66.133:8861
      group: SEATA_GROUP
      data-id: "seataServer.properties"
  data-source-proxy-mode: XA

```

添加注解

在需要分布式事务的方法添加全局事务注解

```

@Override
@Transactional(rollbackFor = Exception.class)
public void saveOrder(Order order) {
    //保存订单
    save(order);

    //扣减用户余额
    userClient.deduct(order.getUserId(), order.getPrice() * order.getNum());

    int i = 100/0; //异常发生
}

```

运行效果

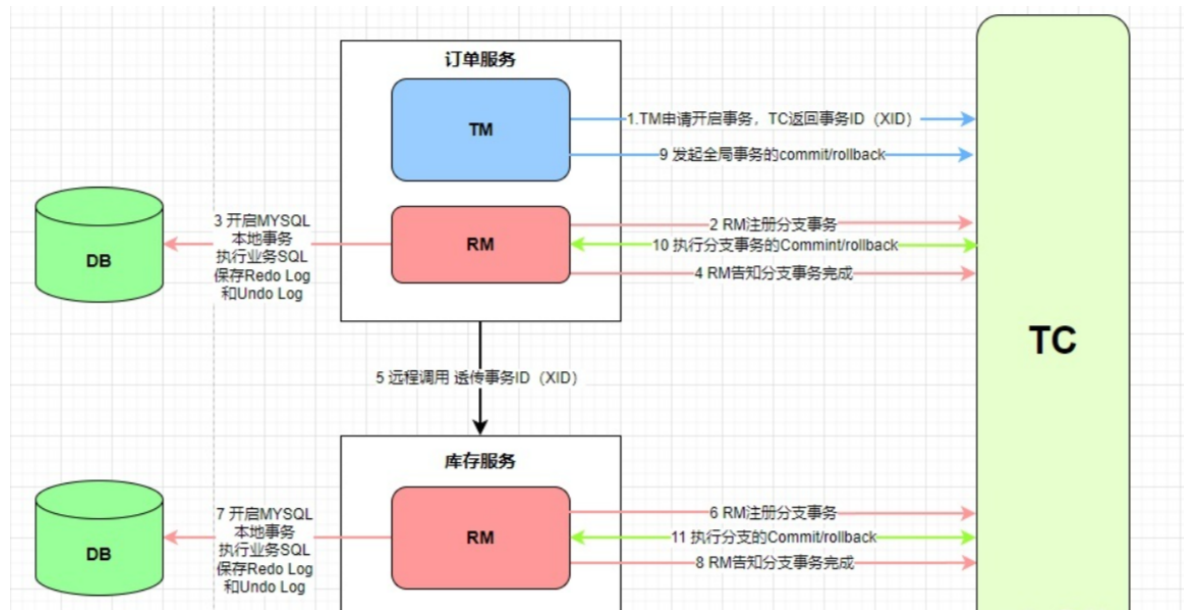
```

per.UserMapper.updateById : <== Updates: 1
le branch rollback process:xid=192.168.66.133:8091:99409899226652679,branchId=99409899226652681,branchType=XA,resourceId=192.168.66.133:8091:99409899226652679-99409899226652681 jdbc:mysql://192.168.66.133:3306/mafeng_user
Rollbacking: 192.168.66.133:8091:99409899226652679-99409899226652681 was rollbacked
Rollbacked result: PhaseTwo_Rollbacked
t in Zipkin's URL [http://192.168.66.133:9411] is provided - that means that load balancing will not take place
ST http://192.168.66.133:9411/api/v2/spans

```

7、Seata分布式（二）AT模式

Seata的AT模式原理

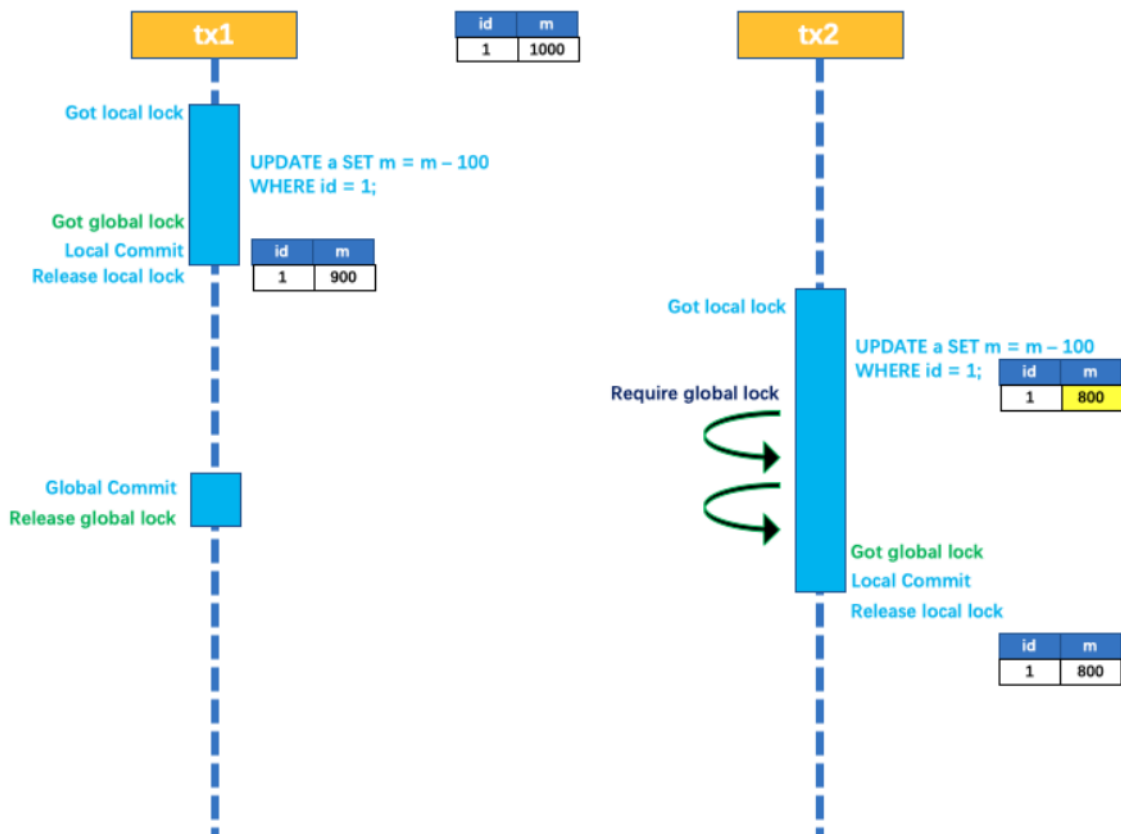


AT模式的写隔离

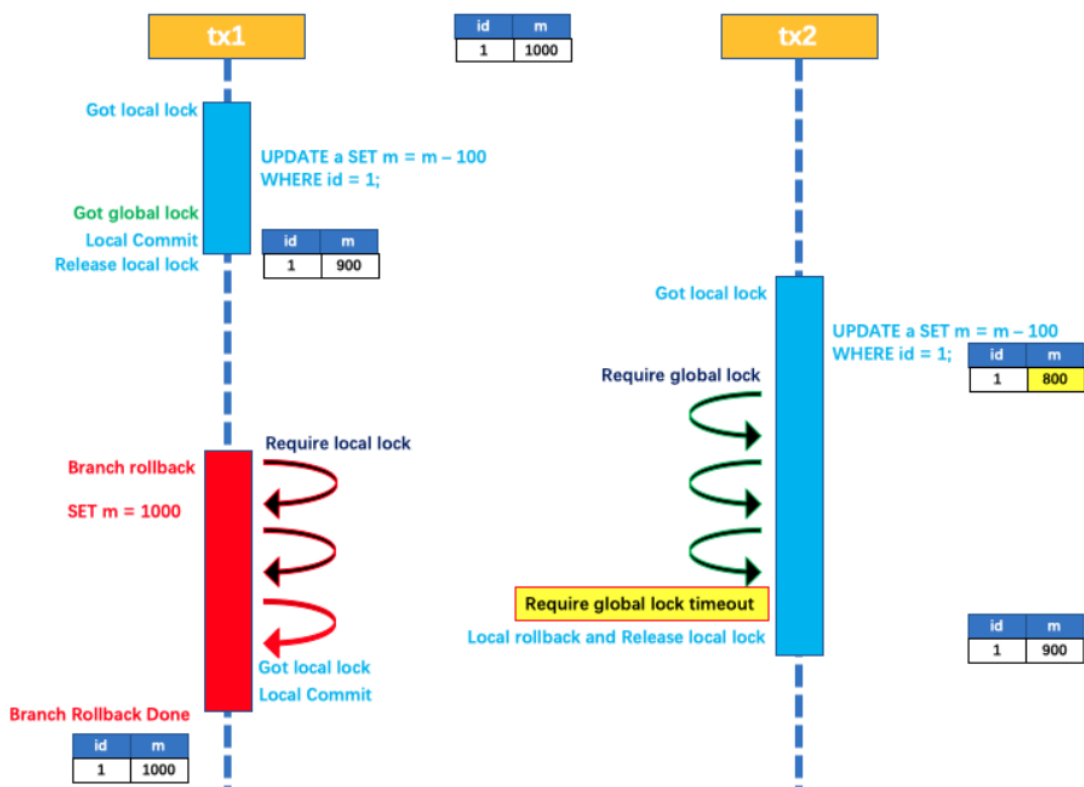
- 一阶段本地事务提交前，需要确保先拿到 **全局锁**。
- 拿不到 **全局锁**，不能提交本地事务。
- 拿 **全局锁** 的尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁。

两个全局事务 tx1 和 tx2，分别对 a 表的 m 字段进行更新操作，m 的初始值 1000。

tx1 先开始，开启本地事务，拿到本地锁，更新操作 $m = 1000 - 100 = 900$ 。本地事务提交前，先拿到该记录的 **全局锁**，本地提交释放本地锁。tx2 后开始，开启本地事务，拿到本地锁，更新操作 $m = 900 - 100 = 800$ 。本地事务提交前，尝试拿该记录的 **全局锁**，tx1 全局提交前，该记录的全局锁被 tx1 持有，tx2 需要重试等待 **全局锁**。



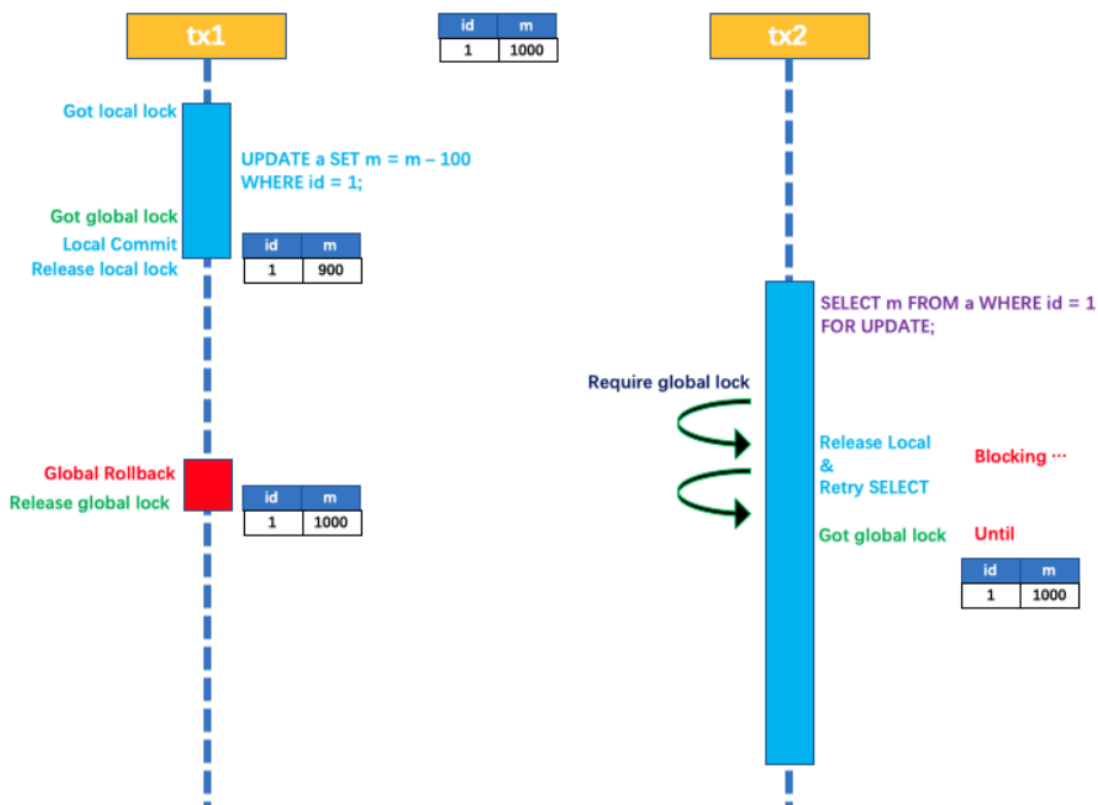
tx1 二阶段全局提交，释放 **全局锁**。tx2 拿到 **全局锁** 提交本地事务。



AT模式的读隔离

在数据库本地事务隔离级别 **读已提交 (Read Committed)** 或以上的基础上，Seata (AT 模式) 的默认全局隔离级别是 **读未提交 (Read Uncommitted)**。

如果应用在特定场景下，必需要求全局的 **读已提交**，目前 Seata 的方式是通过 SELECT FOR UPDATE 语句的代理。



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。

Seata的AT模式步骤

创建undo_log表

在mafeng_order和mafeng_user库添加undo_log表

```
-- for AT mode you must to init this sql for you business database. the seata
server not need it.
CREATE TABLE IF NOT EXISTS `undo_log`
(
    `branch_id`      BIGINT(20)   NOT NULL COMMENT 'branch transaction id',
    `xid`            VARCHAR(100) NOT NULL COMMENT 'global transaction id',
    `context`        VARCHAR(128) NOT NULL COMMENT 'undo_log context,such as
serialization',
    `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 KEY `ux_undo_log` (`xid`, `branch_id`)
) ENGINE = InnoDB
  AUTO_INCREMENT = 1
  DEFAULT CHARSET = utf8mb4 COMMENT ='AT transaction mode undo table';
```

修改配置

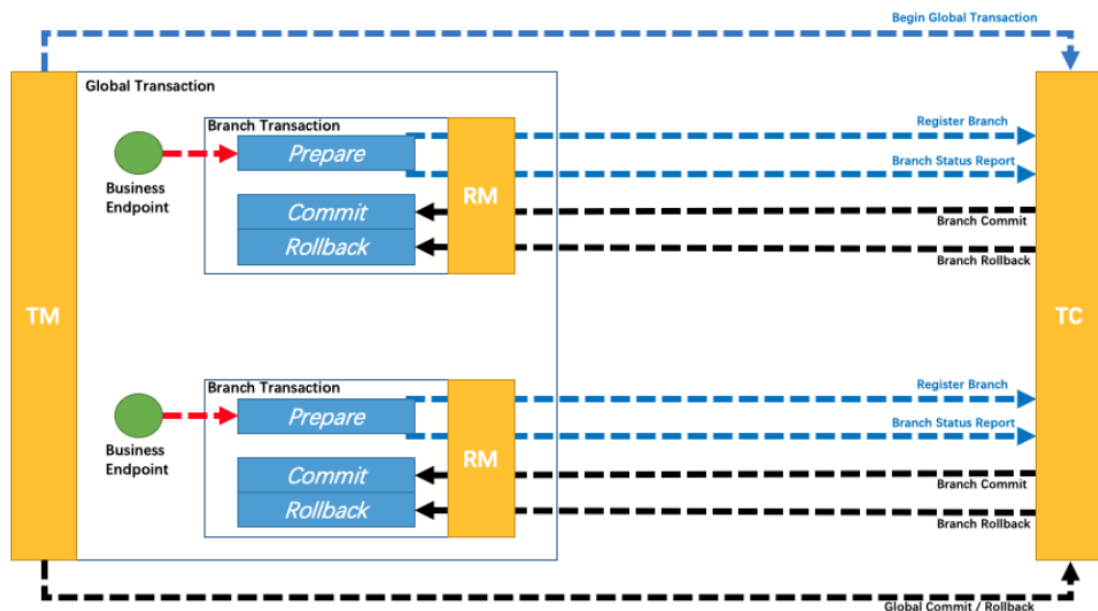
```
seata:
  enable: true
  application-id: seata-cilent-order
  tx-service-group: default_tx_group
  registry:
    type: nacos
    nacos:
      application: seata-server # seata 服务名
      # 非本地请修改具体的地址
      server-addr: http://192.168.66.133:8861
      group: SEATA_GROUP
  config:
    type: nacos
    nacos:
      # nacos ip地址
      server-addr: http://192.168.66.133:8861
      group: SEATA_GROUP
      data-id: "seataServer.properties"
      data-source-proxy-mode: AT
```

运行效果

```
it.config.impl.CacheData : [fixed-192.168.66.133:8861] [add-listener] ok, tenant=, dataId=client.rm.lock.retryTim
it.config.impl.CacheData : [fixed-192.168.66.133:8861] [add-listener] ok, tenant=, dataId=client.rm.lock.retryTim
.e branch rollback process:xid=192.168.66.133:8091:99409899226652694,branchId=99409899226652694,branchType=AT,resou
rollbacking: 192.168.66.133:8091:99409899226652694 99409899226652696 jdbc:mysql://192.168.66.133:3306/mafeng_user
168.66.133:8091:99409899226652694 branch 99409899226652696, undo_log deleted with GlobalFinished
rollbacked result: PhaseTwo_Rollbacked
: in Zipkin's URL [http://192.168.66.133:9411] is provided - that means that load balancing will not take place
IT http://192.168.66.133:9411/ani/v2/spans
```

8、Seata分布式（三）TCC模式

Seata的TCC模式原理



TCC模式与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复。需要实现三个方法：

- Try：资源的检测和预留；
- Confirm：完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。
- Cancel：预留资源释放，可以理解为try的反向操作。

Seata的TCC模式步骤

UserController

```
/**
 * 扣减账户余额
 */
@PutMapping("/deduct/{id}/{money}")
public void deduct(@PathVariable("id") Long id,@PathVariable("money") Long money){
    //userService.deduct(id,money);
    userService.deductTCC(id,money);
}
```

UserService接口

```
@LocalTCC
public interface UserService extends IService<User> {

    public void queryUserByName();

    void deduct(Long id, Long money);

    @TwoPhaseBusinessAction(name = "deductTCC",commitMethod =
"commitTCC",rollbackMethod = "rollbackTCC")
    void deductTCC(@BusinessActionContextParameter("id") Long id,
@BusinessActionContextParameter("money") Long money);
}
```

```

    public boolean commitTCC(BusinessActionContext actionContext);

    public boolean rollbackTCC(BusinessActionContext actionContext);
}

```

UserServiceImpl实现

```

@Service
@Slf4j
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    @Override
    @SentinelResource("queryUserByName")
    public void queryUserByName() {
        System.out.println("根据name查询用户");
    }

    @Override
    @Transactional(rollbackFor = Exception.class)
    public void deduct(Long id, Long money) {
        User user = getById(id);
        user.setBalance(user.getBalance()-money);
        updateById(user);
    }

    //用于锁定第一阶段资源
    private final Map<String,User> userMap = new ConcurrentHashMap<>();

    @Override
    public void deductTCC(Long id, Long money) {
        log.info("执行第一阶段");
        User oldUser = getById(id);
        userMap.put(RootContext.getXID(),oldUser);
    }

    @Override
    public boolean commitTCC(BusinessActionContext actionContext) {
        log.info("执行第二阶段commit");
        Integer id = (Integer)actionContext.getActionContext("id");
        Integer money = (Integer)actionContext.getActionContext("money");
        //执行业务
        User user = getById(id);
        user.setBalance(user.getBalance()-money);
        boolean flag = updateById(user);
        return flag;
    }

    @Override
    public boolean rollbackTCC(BusinessActionContext actionContext) {
        log.info("执行第二阶段rollback");
        User user = userMap.get(actionContext.getXid());
        boolean flag = updateById(user);
        return flag;
    }
}

```


运行效果

```
processor : rm handle branch rollback process:xid=192.168.66.133:8091:99409899226652712,branchId=99409899226652714
           : Branch Rollbacking: 192.168.66.133:8091:99409899226652712 99409899226652714 deductTCC
viceImpl : 执行第二阶段rollback
updateById : ==> Preparing: UPDATE tb_user SET username=?, sex=?, balance=? WHERE id=?
updateById : ==> Parameters: 张三(String), 男(String), 180(Long), 3(Long)
updateById : <== Updates: 1
rager      : TCC resource rollback result : true, xid: 192.168.66.133:8091:99409899226652712, branchId: 99409899226
           : Branch Rollbacked result: PhaseTwo_Rollbacked
figuration : The port in Zipkin's URL [http://192.168.66.133:9411] is provided - that means that load balancing wil
           : HTTP POST http://192.168.66.133:9411/api/v2/trace
```

9、Seata分布式（四）SAGA模式

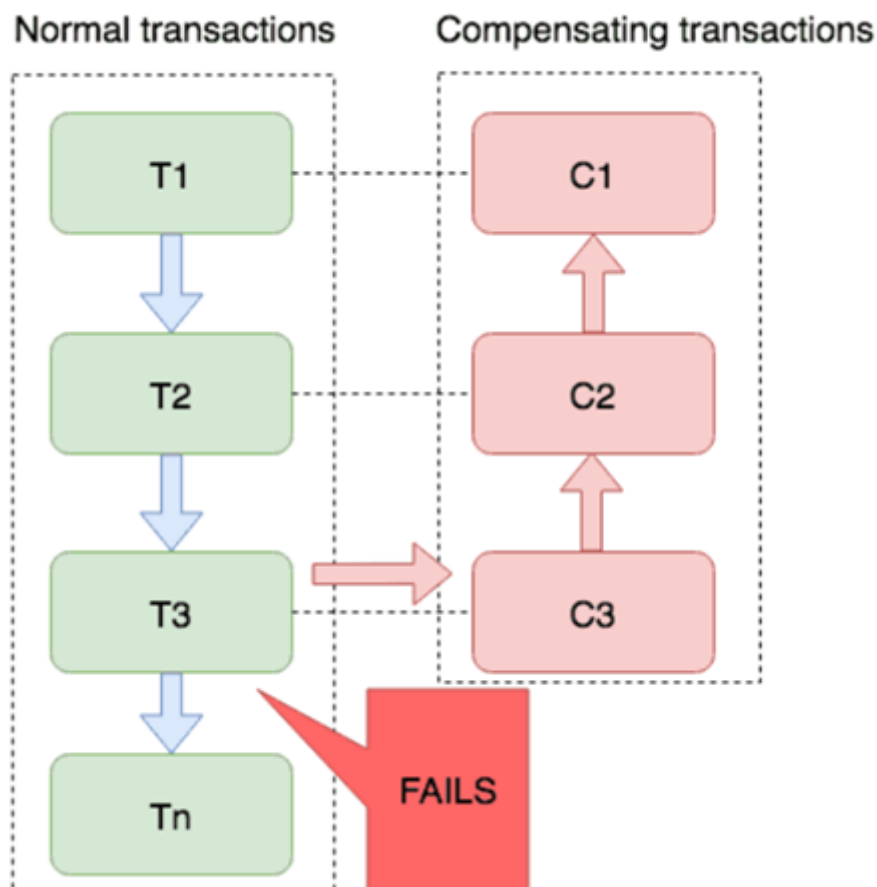
Saga 模式是 Seata 即将开源的长事务解决方案，将由蚂蚁金服主要贡献。

其理论基础是Hector & Kenneth 在1987年发表的论文Sagas。

Seata官网对于Saga的指南：<https://seata.io/zh-cn/docs/user/saga.html>

在 Saga 模式下，分布式事务内有多参与者，每一个参与者都是一个冲正补偿服务，需要用户根据业务场景实现其正向操作和逆向回滚操作。

分布式事务执行过程中，依次执行各参与者的正向操作，如果所有正向操作均执行成功，那么分布式事务提交。如果任何一个正向操作执行失败，那么分布式事务会去退回去执行前面各参与者的逆向回滚操作，回滚已提交的参与者，使分布式事务回到初始状态。



Saga也分为两个阶段：

- 一阶段：直接提交本地事务

- 二阶段：成功则什么都不做；失败则通过编写补偿业务来回滚

适用场景：

- 业务流程长、业务流程多
- 参与者包含第三方公司或遗留系统服务，无法提供 TCC 模式要求的三个接口
- 典型业务系统：如金融网络（与外部金融机构对接）、互联网微贷、渠道整合等业务系统

10、总结Seata四种模式

	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	<ul style="list-style-type: none"> • 对性能要求较高的事务。 • 有非关系型数据库要参与的事务。 	<ul style="list-style-type: none"> • 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