## 分布式事务Seata: RPC协议

简介：

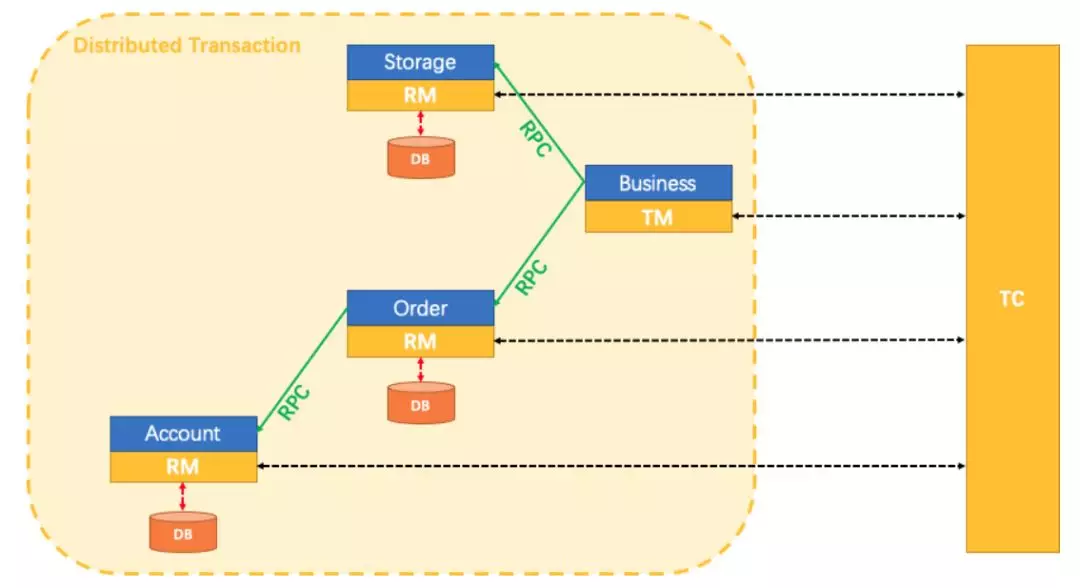
★★ 源码：https://zhuanlan.zhihu.com/p/87100741

### 历史 阿里14年起 19年改

1. TXC：Taobao Transaction Constructor，阿里巴巴中间件团队自 2014  
   年起启动该项目，以满足应用程序架构从单一服务变为微服务所导致的分布式事务问题。
2. GTS：Global Transaction Service，2016 年 TXC 作为阿里中间件的产品，更名为 GTS 发布。
3. FESCAR：2019 年开始基于 TXC/GTS 开源 FESCAR。
4. 后更名为Seata

社区里也有一些开源的分布式解决方案的框架，比如ByteTCC、LCN，但是这些框架没有一个权威的组织在维护

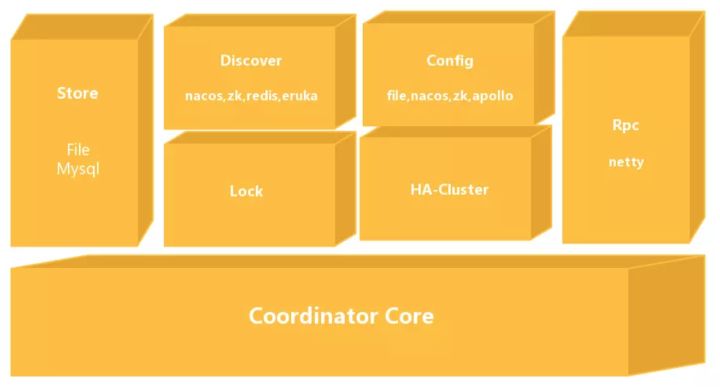
* Transaction Coordinator(TC)：维护全局和分支事务的状态，驱动全局事务提交与回滚。
* Transaction Manager™：定义全局事务的范围：开始、提交或回滚全局事务。
* Resource Manager(RM)：管理分支事务处理的资源，与 TC通信以注册分支事务并报告分支事务的状态，并驱动分支事务提交或回滚。



### Transaction Coordinator (TC)：

事务协调器，维护全局事务的运行状态，负责协调并驱动全局事务的提交或回滚。

整体的模块图如上所示:



#### Coordinator Core:

在最下面的模块是事务协调器核心代码，主要用来处理事务协调的逻辑，如分支的注册, commit, rollback 等协调活动。

当 TC 启动时, 先恢复本机的 Session, 然后启动 RPC Server, 最后注册自己的地址到注册中心, 这些我们前面已经介绍过了, 除此之外, TC 还会启动几个后台线程, 这些线程保证了 TC 的协调工作能够在发生错误时, 最终能顺利完成。

后台任务分别是回滚重试, 提交重试, 异步提交, 超时检测, 删除没用的 AT 模式 undo log。

#### Discover: 服务注册/发现模块，用于将 Server 地址暴露给我们 Client。

订阅的过程如下: 1. 获取当前 clusterName 名字 2. 判断当前 cluster 是否已经获取过了，如果获取过就从map中取。 3. 从 Redis 拿到地址数据，将其转换成我们所需要的数据。 4. 将数据变动的 Listener 注册到 Redis

我们知道 Server 是将自己注册到注册中心, 然后 Client 订阅更新, 并得到 Server 的列表, 最后通过负载均衡选择一个 Server 进行连接。当连接建立成功后, Server 会保存所有的连接, 在需要进行分支回滚和提交时, 从所有 RM 的连接记录中, 找到对应 RM 的所有连接, 它会首先寻找最原始的 RM 节点, 如果该节点宕机了, 它会找到该 RM 的其他节点, 然后发送分支提交请求。

#### Config: 只有随机和轮训,用来存储和查找我们服务端的配置。

getShort/getInt/Long/Boolean/Config()：通过dataId来获取对应的值。

putConfig：用于添加配置。

removeConfig：删除一个配置。

add/remove/get ConfigListener：添加/删除/获取 配置监听器，一般用来监听配置的变更。

目前为止有四种方式获取 Config：File(文件获取), Nacos, Apollo, ZK，etcd。在 Seata 中首先现在项目 resources 下保存一个 registry.conf 文件，在该文件中配置具体使用 Config 接口哪个实现类。

#### Lock: 读已提交隔离，锁模块，用于给 Seata 提供全局锁的功能。

Seata 中可以保证写操作的互斥性，而读的隔离级别一般是读未提交，但是提供了达到读已提交隔离的手段。

* acquireLock：用于对我们的 BranchSession 加锁，这里虽然是传的分支事务 Session，实际上是对分支事务操作的数据行加锁，成功返回 true。
* isLockable：根据事务 ID，资源 ID，锁住的Key来查询是否已经加锁。
* releaseLock: 释放分支事务的所有锁。
* releaseGlobalSessionLock： 释放全局事务的所有分支事务的锁。
* cleanAllLocks：清除所有的锁。

一种是基于内存的锁(Session 存储模式为 File 时使用), 一种是基于 DB 的(Session 存储模式为 DB 时使用)，它们都实现了 Locker 接口

#### Store: 存储模块，用来将我们的数据持久化，防止重启或者宕机数据丢失。

RM 可能会发生频繁回滚那么其完全无法应对高并发的场景。

在 Seata 中默认提供了文件方式的存储，下面我们定义我们存储的数据为 Session，而我们的TM创造的全局事务数据叫 GlobalSession，RM 创造的分支事务叫 BranchSession，一个 GlobalSession 可以拥有多个 BranchSession。我们的目的就是要将这么多 Session 存储下来。

Seata 中目前有 2 种实现方案, 一种是基于文件的, 一种是基于 DB 的

##### File：

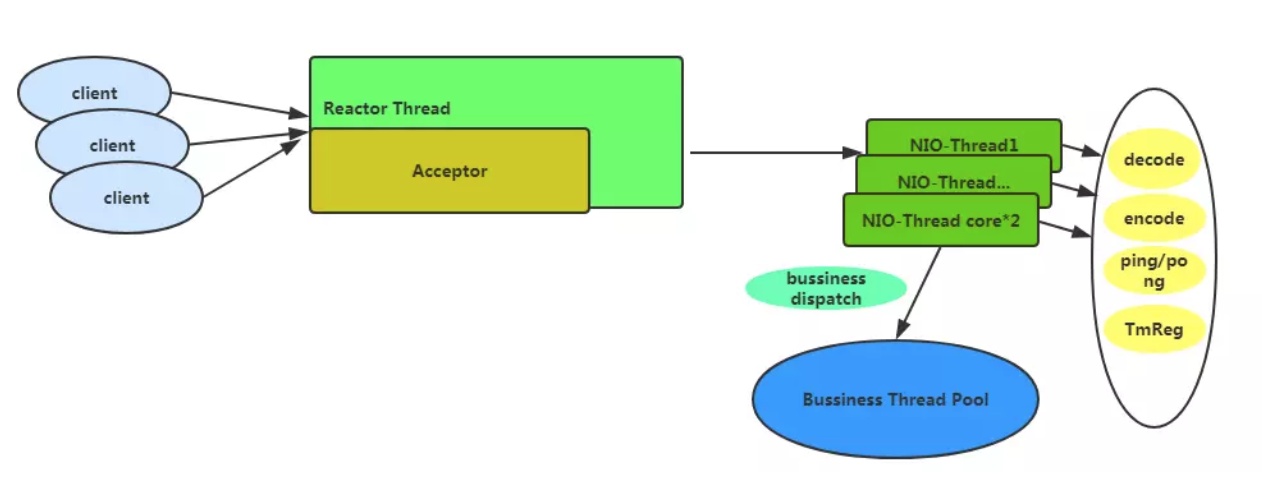
基于文件的实现是 FileTransactionStoreManager, 它可以使用同步刷盘或异步刷盘的策略，每当有 Session 的状态的更新时，它都会将变化的内容存储起来。为了防止存储文件的无限增殖，当达到一定条件时，它会另打开一个文件从头开始记录，并将之前的文件保存起来。这里有一个非常巧妙的设计，就是该方案既能保证所有超时事务不丢，只有已完成的事务被清除，同时文件的大小也得到了控制。

如果 TC 宕机，重启时只要先读取 historyFullFile，再读取 currentDataFile 就能恢复所有数据。

##### db：

基于 DB 的实现相较于基于文件的实现就显得朴实无华，logStore 实际上就是一个 DAO 层的接口，对应了数据的 CRUD，在重启恢复时只不过是按照条件遍历 DB 中的所有数据，进行 Session 恢复。

#### Rpc: 用于和其他端通信。



如果采用默认的基本配置, 那么会有一个 Acceptor 线程用于处理客户端的链接，会有 cpu\*2 数量的 NIO-Thread，在这些 NIO-Thread 线程中不会做业务太重的事情，只会做一些速度比较快的事情，比如编解码，心跳事件，和 TM 注册。一些比较费时间的业务操作将会交给业务线程池，默认情况下业务线程池配置为最小线程为 100，最大为 500。

**关于 Netty 的使用基础, 我们这里就不详细介绍了, 简单说就是对于每个连接都会绑定上数据的 handler, 它会按照责任链的原则, 顺着 handler 的绑定顺序, 处理数据, 这里简单看下它都绑定了什么 handler:**

#### HA-Cluster: 高可用集群，目前还没开源。

为 Seata 提供可靠的高可用功能。

### Transaction Manager (TM)

首先 TM 在启动的时候会去连接 TC Server, 然后然后通过该 TM Client 与 TC 模块进行通讯。在 TM 模块中最核心的接口就是 GlobalTransaction, 里面包含了全局事务的创建, 提交, 回滚过程, 其实质就是向 TC 发送 RPC 请求。

初始化：

#### TransactionManager：

它调用 transactionManager 发送消息, 然后将涉及到的全局事务 XID 保存起来.

TransactionManager 才是真正做实事的, 消息的发送工作都在这里完成。好了, 至此我们知道了哪个接口管理着全局事务的记录, 哪个接口真正进行 RPC 调用, 那么谁才是这些接口的真正调用者呢? Seata 使用了模板方法模式来进行这部分工作

#### TransactionalTemplate：模板方法：

1. 看看当前是不是已经在一个分布式事务中了, 如果是, 则复用现存的全局事务, 否则创建新的 - 什么时候会出现已经存在全局事务的情况呢? 假设 A 调用了 B, A 创建了全局事务 GT1, B 碰巧也执行了上述的模板, 这时候 B 就不会创建新的全局事务, 而是使用 GT1, 这实际上是前面提到的事物的传播

2. 如果是自己创建的全局事务, 则发 RPC 开始事务, 如果不是自己创建的则什么都不干 3. 执行真正的业务逻辑

4. 如果发生了异常, 如果自己创建全局事务, 才负责回滚, 否则就只管异常外抛

5. 如果没发生异常, 如果自己创建全局事务, 才负责提交, 否则就什么都不做

6. 清理工作

#### GlobalTransactional切面

#### TM 怎么将事务信息是怎么传递

实际上 Seata 基于 Spring 切面, 已经帮我们做了这些事, 我们只需要使用 GlobalTransactional 注解就够了,

那么还有一个问题, 事务信息是怎么传递的呢, TM 怎么将全局事务 XID 传递给 RPC 的提供者的呢? 这部分, 根据 RPC 框架的不同, 需要不同的实现, 但是本质上都是一样的, 拦截 RPC 的调用过程, 在 RPC 请求中加一个隐藏属性来存储 XID, RPC 的提供方从请求中获取到该隐藏属性, 然后存储在事务 Context 的 ThreadLocal 中, 我们以

#### Dubbo 为例

Dubbo 是通过一个 Filter 的概念来表示 AOP 特性的，Filter 的注入基于 Dubbo 的 SPI, 而 Seata 这里所做的就是实现了一个 Dubbo Filter, 把事务 Context 和 RPCContext 中的数据做一下绑定。其他 RPC 框架的支持方案基本类似,

官方demo：

### Seata 管理分布式事务的典型生命周期：

* TM 要求 TC 开始新的全局事务，TC 生成表示全局事务的 XID。
* XID 通过微服务的调用链传播。XID 通过微服务的调用链传播。
* RM 在 TC 中将本地事务注册为 XID 的相应全局事务的分支。RM 在 TC 中将本地事务注册为 XID 的相应全局事务的分支。
* TM 要求 TC 提交或回滚 XID 的相应全局事务。TM 要求 TC 提交或回滚 XID 的相应全局事务。
* TC 驱动 XID 的相应全局事务下的所有分支事务，完成分支提交或回滚。TC 驱动 XID 的相应全局事务下的所有分支事务，完成分支提交或回滚。

### AT事务模式:集团的 TXC==事务模式

★★★：<https://chenjiayang.me/2019/05/02/seata-tcc/>

Seata 是阿里近期开源的分布式事务框架，地址：<https://github.com/seata/seata>。框架包括了集团的 TXC（云版本叫 GTS）, TXC 在 Seata 中又叫 AT 模式意为补偿方法是框架自动生成的，对用户完全屏蔽，用户可以向使用本地事务那样使用分布式事务，缺点是仅支持关系型数据库（目前支持 MySQL）,引入 Seata AT 的服务需要本地建表存储 rollback\_info，隔离级别默认 RU 适用场景有限。

### TCC事务模式:蚂蚁金服的 TCC（Try-Confirm-Cancel）

<https://www.bytesoft.org/tcc-intro/>

蚂蚁金服向 Seata 贡献了自己的 TCC 实现，据说已经演化了十多年，大量应用在在金融、交易、仓储等领域。

#### TCC事务机制简介

关于TCC（Try-Confirm-Cancel）的概念，最早是由Pat Helland于2007年发表的一篇名为《Life beyond Distributed Transactions:an Apostate’s Opinion》的论文提出。在该论文中，TCC还是以Tentative-Confirmation-Cancellation作为名称；正式以Try-Confirm-Cancel作为名称的，可能是Atomikos（Gregor Hohpe所著书籍《Enterprise Integration Patterns》中收录了关于TCC的介绍，提到了Atomikos的Try-Confirm-Cancel，并认为二者是相似的概念）。

国内最早关于TCC的报道，应该是InfoQ上对阿里程立博士的一篇采访。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事务机制相对于传统事务机制（X/Open XA Two-Phase-Commit），其特征在于它不依赖资源管理器(RM)对XA的支持，而是通过对（由业务系统提供的）业务逻辑的调度来实现分布式事务。

对于业务系统中一个特定的业务逻辑S，其对外提供服务时，必须接受一些不确定性，即对业务逻辑执行的一次调用仅是一个临时性操作，调用它的消费方服务M保留了后续的取消权。如果M认为全局事务应该rollback，它会要求取消之前的临时性操作，这将对应S的一个取消操作；而当M认为全局事务应该commit时，它会放弃之前临时性操作的取消权，这对应S的一个确认操作。

每一个初步操作，最终都会被确认或取消。因此，针对一个具体的业务服务，TCC事务机制需要业务系统提供三段业务逻辑：初步操作Try、确认操作Confirm、取消操作Cancel。

#### 初步操作（Try）

TCC事务机制中的业务逻辑（Try），从执行阶段来看，与传统事务机制中业务逻辑相同。但从业务角度来看，却不一样。TCC机制中的Try仅是一个初步操作，它和后续的确认一起才能真正构成一个完整的业务逻辑。可以认为

|  |  |
| --- | --- |
| 1 | [传统事务机制]的业务逻辑 = [TCC事务机制]的初步操作（Try） + [TCC事务机制]的确认逻辑（Confirm）。 |

TCC机制将传统事务机制中的业务逻辑一分为二，拆分后保留的部分即为初步操作（Try）；而分离出的部分即为确认操作（Confirm），被延迟到事务提交阶段执行。  
TCC事务机制以初步操作（Try）为中心的，确认操作（Confirm）和取消操作（Cancel）都是围绕初步操作（Try）而展开。因此，Try阶段中的操作，其保障性是最好的，即使失败，仍然有取消操作（Cancel）可以将其不良影响进行回撤。

#### 确认操作（Confirm）

确认操作（Confirm）是对初步操作（Try）的一个补充。当TCC事务管理器决定commit全局事务时，就会逐个执行初步操作（Try）指定的确认操作（Confirm），将初步操作（Try）未完成的事项最终完成。

#### 取消操作（Cancel）

取消操作（Cancel）是对初步操作（Try）的一个回撤。当TCC事务管理器决定rollback全局事务时，就会逐个执行初步操作（Try）指定的取消操作（Cancel），将初步操作（Try）已完成的事项全部撤回。

在传统事务机制中，业务逻辑的执行和事务的处理，是在不同的阶段由不同的部件来完成的：业务逻辑部分访问资源实现数据存储，其处理是由业务系统负责；事务处理部分通过协调资源管理器以实现事务管理，其处理由事务管理器来负责。二者没有太多交互的地方，所以，传统事务管理器的事务处理逻辑，仅需要着眼于事务完成（commit/rollback）阶段，而不必关注业务执行阶段。

而在TCC事务机制中的业务逻辑处理和事务处理，其关系就错综复杂：业务逻辑（Try/Confirm/Cancel）阶段涉及所参与资源事务的commit/rollback；全局事务commit/rollback时又涉及到业务逻辑（Try/Confirm/Cancel）的执行。其中关系，本站将另行撰文详细介绍，敬请关注！

#### 常用类源码：

Seata 中 TCC 模式的源码并不复杂，主要集中于：

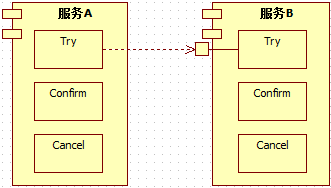
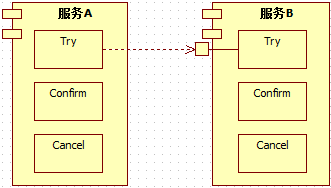
| module | class | 功能 |
| --- | --- | --- |
| seata-spring | GlobalTransactionalInterceptor.class | 全局事务切面逻辑，包括注册全局事务，拿到 xid |
| seata-spring | TccActionInterceptor.class | TCC 参与方切面逻辑 |
| seata-tcc | TCCResourceManager.class | 解析 TCC Bean，保存 TCC Resources，便于后续回调 |
| seata-tcc | ActionInterceptorHandler.class | TCC 分支事务注册实现 |
| seata-server | DefaultCoordinator.class、FileTransactionStoreManager.class | 主要是 TC 的实现、事务存储等实现 |

#### 流程demo：

TCC全局事务必须基于RM本地事务来实现全局事务

TCC服务是由Try/Confirm/Cancel业务构成的，其Try/Confirm/Cancel业务在执行时，会访问资源管理器（Resource Manager，下文简称RM）来存取数据。这些存取操作，必须要参与RM本地事务，以使其更改的数据要么全部commit，要么全部rollback。

这一点不难理解，考虑一下如下场景：



假设图中的服务B没有基于RM本地事务（以RDBS为例，可通过设置auto-commit为true来模拟），那么一旦[B:Try]操作中途执行出错，TCC事务框架后续决定回滚全局事务时，该[B:Cancel]则需要判断[B:Try]中哪些操作已经写到DB、哪些操作还没有写到DB：如果[B:Try]业务有5个写库操作，[B:Cancel]业务则需要逐个判断这5个操作是否生效，并将生效的操作执行反向操作。

不幸的是，由于[B:Cancel]业务也有n（0<=n<=5）个反向的写库操作，此时一旦[B:Cancel]也中途出错，则后续的[B:Cancel]执行任务更加繁重。因为，相比第一次[B:Cancel]操作，后续的[B:Cancel]操作还需要判断先前的[B:Cancel]操作的n（0<=n<=5）个写库中哪几个已经执行、哪几个还没有执行，这就涉及到了幂等性问题。

然而，对幂等性的保障，很可能也需要涉及额外的写库操作，该写库操作又会因为没有RM本地事务的支持而存在类似问题。。。

可想而知，如果不基于RM本地事务，TCC事务框架是无法有效的管理TCC全局事务的

demo：

### 单纯的存储层 2PC

### demo-tcc：

#### Seata TCC 参与方

Seata 中的 TCC 模式要求 TCC 服务的参与方在接口上增加 @TwoPhaseBusinessAction 注解，注明 TCC 接口的名称（全局唯一），TCC 接口的 confirm 和 cancel 方法的名称，用于后续框架反射调用，下面是一个 TCC 接口的案例：

public interface TccAction {

@TwoPhaseBusinessAction(name = "yourTccActionName", commitMethod = "confirm", rollbackMethod = "cancel")

public boolean try(BusinessActionContext businessActionContext, int a, int b);

public boolean confirm(BusinessActionContext businessActionContext);

public boolean cancel(BusinessActionContext businessActionContext);

}