XTS 支付宝分布式事务学习指南

@潇桐 @柳成

支付宝-成都应用研发中心-创新支付工具组-卡券平台

V1.0.8

版本号	修订人	内容提要	修订日期
1.0.0	@潇桐	初建文档，完成 XTS 实例分析和配置部分编写	2014-07-22
1.0.1	@柳成	增加 XTS 理论基础部分	2014-07-23
1.0.2	@柳成	1. 增加 XTS 源码浅析部分 2. 完成排版	2014-07-24
1.0.3	@潇桐	对文档细节提出疑问和修订，给出修改意见	2014-07-25
1.0.4	@柳成	针对修改意见完成细节修改	2014-07-25
1.0.5	@柳成	修正 2.4.2 中几处错误，感谢@宾雨 指正	2014-07-29
1.0.6	@柳成	对一种 recover 的特殊情况进行考虑，总结出答案并添加在 4.4，感谢@宾雨 的讨论	2014-07-30
1.0.7	@柳成	重新绘制 2.4.2 中异库模式下 recover 回查情况的箭头指向，并作出说明，感谢@虞卿 指正	2014-07-31
1.0.8	@柳成	重新绘制 2.4.2 中所有图中二阶段的流程	2014-08-30

目　　录

1 概 述

XTS（e**X**tended **T**ransaction **S**ervice）框架[1]，是支付宝的一个极为核心而且复杂的分布式事务技术框架，在支付宝有广泛地使用，主要用于保证在账务、资金等操作的事务一致性，因此 XTS 框架足以称为支付宝分布式事务框架。因为其应用场景和理论知识的复杂性，使得整个框架的配置和理解不那么简单和易学，因此不少新同学在理解 XTS 和配置 XTS 上走了很多弯路。本文是作者在学习 XTS 的过程中思考和总结的结果，主要对 XTS 的理论基础和基本概念（@柳成 整理）、XTS 的实例分析和配置使用方法（@潇桐 整理）、XTS 源码浅析（@柳成 整理）这三个方面来对 XTS 进行一个较为全面的入门学习，希望能在某些方面能够解答新人学习时的疑惑。

2 XTS 分布式事务原理

@柳成

XTS 分布式事务框架涉及了较多的分布式事务基本概念，但是最主要的概念是**两阶段提交协议(Two Phase Commit)**。本章将以它为目标阐述 XTS 用到的一些基本模型和理论，先对基础概念进行阐述，然后再详细描述 XTS 发起方/参与者是如何构成整个 XTS 分布式事务的流程和逻辑，以及总结一下关于发起者/参与者在几种模式下的不同行为。

2.1 分布式事务

事务的基本概念以及 **ACID 原则**：请看@潇桐 的文章[2]：点这里↗

分布式事务的概念[3]：事务的参与者分布于网络环境中的不同的节点。也就是说可以将多个事务资源纳入到一个单一的事务之中，并且这些事务资源可以分布到不同的机器上。这些承载分布式资源的机器可能是出于同一个网络中，也可能处于不同的网络中。甚至说，某个事务资源本质上就是一个通过 HTTP 访问的单纯的 Internet 资源。因此一个分布式事务的服务操作可能会访问不止一个事务资源（比如访问两个不同的数据库服务器），也可能调用另一个服务。

更多关于分布式事务的概念请自行 Google。

2.2 为什么需要 XTS

传统事务是使用数据库自身的事务属性，而数据库自身的事务属性是局限于当前实例，不能实现跨库事务，而对于大型的分布式系统，不仅有跨库事务的存在，还存在不同系统之间的 RPC 调用，这种调用往往也需要保证一致性，因此就需要一种事务框架协调整个调用链路里的应用和数据库以保证一致性，因此需要一个分布式事务框架来统一管理和协调一个事务中原子活动，XTS 就应运而生了。但是，为何不用现有的分布式事务框架而需要自己写一个呢？可以参考 XTS 创建之初@鲁肃 的分析*以及设计草案[4]。主要是由于需要**结合支付宝自身的业务特点**，需要寻找可信技术，并且开源框架的维护成本和实现难度不符合要求，因此需要自己实现。

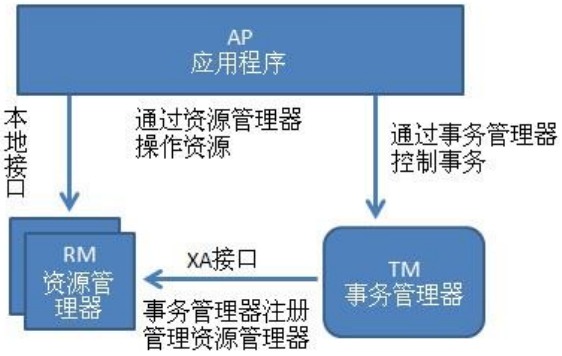
2.3 支付宝分布式事务基础模型

由于在支付宝 SOA 架构下，一次业务请求将会跨多个服务，多个服务协同完成一次业务时会由于业务约束或者系统故障造成多个系统中数据不一致或者通信延迟等问题，因此支付宝分布式事务必须要保证多个系统间的 ACID。本节主要概述支付宝分布式事务的原理，也是 XTS 框架实现的基本原理。

*建议反复学习“SOA 化金融系统的分布式事务.ppt”，ppt 内容有点抽象，可能头一次看不太明白，但是 XTS 的设计思想和原理都在 ppt 中，反复学习和思考会有收获。

2.3.1 X/Open DTP 模型

支付宝的基本分布式事务模型采用 X/Open DTP(Distributed Transaction Processing Reference Model)，模型如下：



其中：

AP(Application Program)：也就是应用程序，可以理解为使用 DTP 的程序；

RM(Resource Manager)：资源管理器，这里可以是一个 DBMS，或者消息服务器管理系统，应用程序通过资源管理器对资源进行控制，资源必须实现 XA 定义的接口；

TM(Transaction Manager)：事务管理器，负责协调和管理事务，提供给 AP 应用程序编程接口以及管理资源管理器。

AP 可以和 TM 以及 RM 通信，TM 和 RM 互相之间可以通信，TX 接口用于应用程序向事务管理器发起事务、提交事务和回滚事务，DTP 模型里面定义了 XA 接口；TM 和 RM 通过 XA 接口进行双向通信，例如:TM 通知 RM 提交事务或者回滚事务，RM 把提交结果通知给 TM。AP 和 RM 之间则通过 RM 提供的 Native API 进行资源控制，这个没有统一的 API 规范，根据框架不同而各自归约。

2.3.2 两阶段提交协议

TM 和 RM 间采取**两阶段提交(Two Phase Commit)**的方案来解决一致性问题：

两阶段提交需要一个**协调者**（这里为 TM）来掌控所有**参与者**节点（这里为各个 RM）的操作结果并且指引这些节点是否需要最终提交。

第一阶段：又称**准备(prepare)阶段**。事务管理者向各个资源管理器发送 prepare 请求，资源管理器在得到请求后做预处理操作，预处理可能是做预检查，也可能是把请求临时存储，可以理解为是一种试探性提交。下面是一阶段具体步骤[5]：

a. 协调者会问所有的参与者结点，是否可以执行提交操作。

b. 各个参与者开始事务执行的准备工作：如：为**资源上锁**，预留资源，写 undo/redo log……

c. 参与者响应协调者，如果事务的准备工作成功，则回应“可以提交”，否则回应“拒绝提交”。

第二阶段：又叫**提交(commit)阶段**。是指事务真正提交或者回滚的阶段。如果事务管理者发现事务参与者有一个在 prepare 阶段出现失败，则会要求所有的参与者进行回滚。如果管理者发现所有的参与者的 prepare 操作都是成功，那么他将向所有参与者发出提交请求，这时所有参与者才会正式提交。由此保证了要么全部成功要么全部失败。下面是具体步骤[5]：

a. 如果所有的参与者都回应“可以提交”，那么协调者向所有的参与者发送“正式提交”的命令。参与者完成正式提交，并释放所有资源，然后回应“完成”，协调者收集各结点的“完成”回应后结束事务。

b. 如果有一个参与者回应“拒绝提交”，那么，协调者向所有的参与者发送“回滚操作”，并释放所有资源，然后回应“回滚完成”，协调者收集各结点的“回滚”回应后，取消这个事务。

下图是两阶段成功和失败的示例图[6]：

- 阶段一：
  - 1. TM向所有RM发送准备请求
  - 2. RM收到准备请求后，进行准备操作，并向TM回复是否可以提交
- 阶段二：
  - 1. TM收到所有TM的回复后，如果所有RM均可以提交，向所有RM发送提交请求，否则发送回滚请求
  - 2. RM根据TM发送的消息控制本地事务提交/回滚

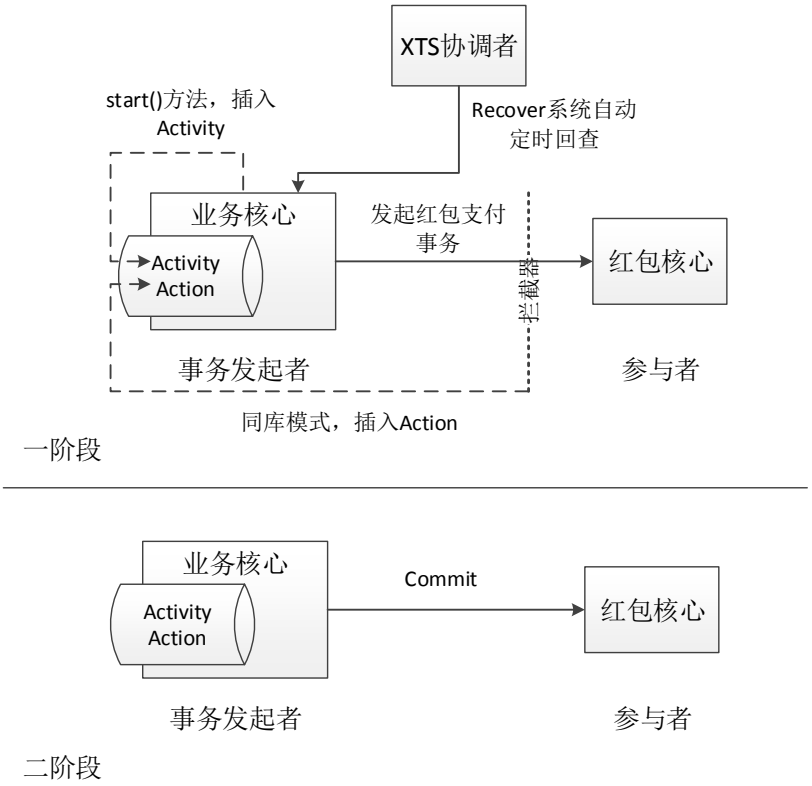
2.3.3 最末参与者优化(LPO)

由于两阶段提交的“准备”操作的实现复杂性和效率都会在实际业务中产生影响，支付宝分布式事务在两阶段提交中引入一个简单的优化——最末参与者优化(Last Participant Optimization)，在这个模型中将有一个参与者不参加两段提交的过程（称为单阶段参与者），而是在其余两阶段参与者都准备好之后，再请求单阶段参与者提交，单阶段参与者的提交结果将决定整个分布式事务的结果。如果单阶段参与者提交成功，那么协调者要求其余参与者提交，如果提交失败则协调者要求其余参与者事务回滚。

在较新版本的 XTS 中，LPO 这个思想已经演变成了另外一种呈现形式：所谓的“单阶段参与者”在 XTS 中扮演**发起方**的角色，它的功能和两阶段提交中的协调者非常相似但又不尽相同，下面将会详细阐述。

2.4 XTS 基本概念和分布式事务执行流程

先上一张图来描述一下在 XTS 框架下，分布式事务大体上是如何执行的：



上图是以红包来举一个简单的例子。图中的“业务核心”可以是任何一个与红包相关的业务系统，比如 paycore，它主要是发起一个分布式事务，需要调用到红包的业务系统，也就是红包核心，完成红包支付等业务活动。XTS 协调者实际上就是 XTS Server，它会统一协调参与分布式事务各个参与者之间提交/回滚操作。关于“同库模式”、“Activity”、“Action”的概念下面紧接着会讲到。

可以非常清楚地看到，**整个 XTS 分布式事务就是完整地执行了两阶段提交协议**。下面对上图几个 XTS 中非常重要的基本概念进行解释。

2.4.1 事务发起者&参与者

XTS 中非常重要的概念，事务发起者又称为**发起方**，它是整个业务活动的主体、是服务的编排者，由它启动业务活动并决定业务活动提交或回滚。简单的例子如上图中，业务核心可以是 paycore 等，它需要调用红包核心完成红包支付的功能，那么此时它是发起分布式事务活动的发起方，它命令其他系统协助完成一次完整的分布式事务。那么参与到这一次分布式事务中来的其他系统就称为**参与者**，这里红包核心就是参与者，它协助发起方完成相应的动作，执行具体的业务逻辑。

两个定义： Activity：把整个分布式事务称为一次业务活动(Activity) Action：把参与者的一次方法调用称为原子活动(Action)

分布式事务必须确保各个角色的一致性和有效性，因此要保证框架进行提交或者回滚，Activity 和 Action

必须在分布式事务执行阶段将状态等信息记录下来，如果这些信息只记录在内存或者本地存储中，容错性低，无法应付宕机，因此这些状态信息记录在 DB 中，通过建立分布式业务控制活动主表(beyond_business_activity)来记录全局事务的活动状态，以及原子业务活动表(beyond_business_action)来记录原子业务活动的状态。在开始一个分布式事务的时候框架先创建 activity 主活动记录，每调用一个参与者就会再创建一个 action 原子活动记录，通过 TX_ID 字段对应。结果就是，XTS 围绕这两个记录存放地点以及两阶段提交流程，衍生了较为复杂的几种模式，各种组合情况各有不同，一一来讲。

对于发起方来说，根据 Activity 记录存放的地点，分为两种情况：

a. 同库模式

发起方会在自己业务库的事务模板中开起分布式事务，**并且 beyond_business_activity 存储在业务库中**，因此同一物理机上不同数据库都不能算同库。适用业务量较大的系统。

优点：对分布式事务活动表的操作都在业务方本地进行，不需要额外的远程调用，速度快，业务方使用简单，不需要写回查。

缺点：需要给框架配置相关的数据源、事务模板、DAO 等；必须确保发起方开启的外部事务于框架是同数据源。

b. 异库模式

发起方会在自己业务库的事务模板中开起分布式事务，而 **beyond_business_activity 和 beyond_business_action 这两张表会存储在 XTS Server 中**，也就是说，这两张表和业务表是在不同的数据库内。现在的异库模式通常要求 RPC 使用 XTS Server 自己的数据库元数据。适用业务量较小的系统。

优点：所有的分布式事务活动都集中管理，管理方便，易于排查问题。

缺点：基于 rpc 速度较同库模式慢，业务方需要写回查。

注：同库模式和异库模式存在的原因是必须记录主事务的信息到 DB 中，方便提交/回滚以及 Recover（如果是同库模式，主要是 recovery，如果是异库模式就是提交/回滚+Recovery），确保各个系统间的一致性，但是由于分布式事务中必然存在一些外围业务系统的接入，而且根据业务情况无法保证数据表总是存放在发起方本地库中，因此产生这两种模式。

对于参与者来说，分为三种情况：

a. local 模式

事务元数据 Action 记录在本地，和 Activity 同库，且没有直接 remote 参与者，发起方在本地对参与者发起提交/回滚。

优点：原子活动框架创建，其存储在本地和 Activity 同库，速度快。

缺点：原子活动的 context 内容由框架获取预处理阶段提交的参数，无法修改；local 模式下无法

创建嵌套的分布式事务（这个说法在版本比较新的 XTS 里，其实不成立，根据 XTS 代码执行的流程，local 模式下的参与者也是可以成为嵌套事务参与者的，只是这种模式并不常见，而且一般在开发时会提前进行约定，这种特殊情况极少出现）。

b. remote 模式

没有配置本地数据源，Activity 和 Action 存储在 XTS Server，整个回滚和提交过程将委托 XTS Server 进行。

优点：自行创建原子活动记录，可以控制 context 的内容；该模式下发起方可以启动嵌套的分布式事务。

缺点：创建原子活动记录需要手动触发，该记录由远程服务存储，效率较低。

c. mix 模式

即事务元数据 Activity 存放在发起方本地库，但是有直接 remote 参与者，可以理解为既有 local 又有 remote 参与者，那么发起方会先调用 XTS Server 提交、回滚 remote 参与者，成功后再调用本地 action 进行提交和回滚。

优缺点：见上，主要是根据业务情况来指定。

注：其实对参与者 3 种情况的理解比较复杂：

参与者所谓的 local 和 remote 模式，在发起方为异库时，无区别，都是将 action 记录在 XTS Server 上；发起方为同库时，local 模式下拦截器会自动将 action 记录到发起方本地库，而 remote 模式下拦截器会将 action 记录到 XTS Server；如果是 mix 模式，主要是说发起方同库，local 会记录 action 到发起方本地，remote 会放入 XTS Server，异库时，就没有 mix 之说，都会把 action 放入到 XTS Server。

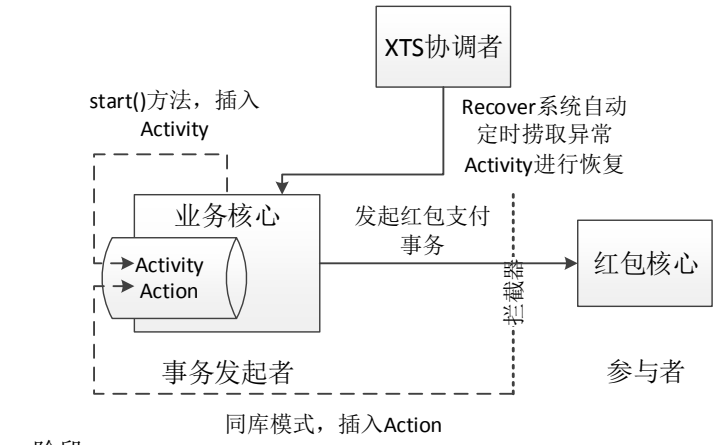
那么 local 模式和 remote 模式到底怎么来的呢？除了由于发起方同库/异库时 action 记录存放地点的不同，还因为在嵌套事务情形下，不存在所谓的 local 模式参与者。什么意思呢？所谓嵌套事务，就是 A->B->C，其中 A 为发起方，调用 B，然后 B 再次调用 C，而不是 A 直接调用 C。这时，拦截器无法拦截 B->C 的过程，无法自动将 action 记录到 A 本地，而且 A 无法在本地展开 C 的动作，因此 C 必须要自己去将 action 记录到 XTS Server 上，当提交或者回滚时，需要 A 向 XTS Server 来发送请求，由 XTS server 提交 C 的 action。

对于嵌套事务是如何发起的，以及嵌套事务下 local 和 remote 参与者的 action 的记录情况可以参考分布式事务启动策略-v4。

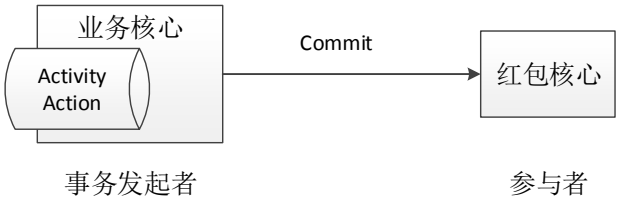
2.4.2 发起方&参与者不同模式下执行流程

发起方有同库/异库两种模式，而参与者有 local/remote/mix 三种模式，又加上嵌套事务的情况，这么多组合情况下，Activity 和 Action 记录到底是记录在哪里？两阶段提交过程中发起方/参与者/XTS Server 见的关系又是什么？这里还是用红包的简单例子来看，用几幅流程图来理清这些关系。

a. 发起方：同库模式&参与者：local



一阶段



二阶段

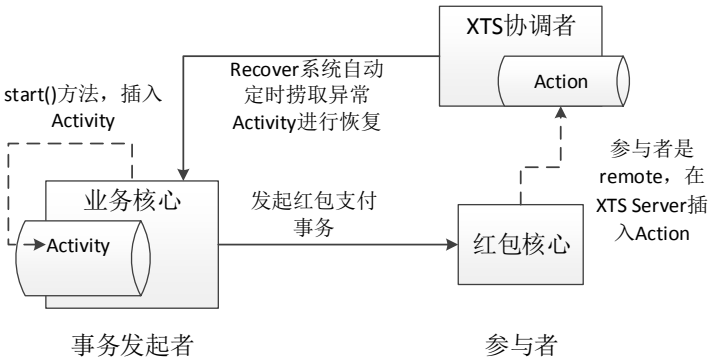
一阶段：

同库模式，Activity 记录在发起方本地，对参与者发起事务时拦截器自动将 action 记录到发起方本地库中

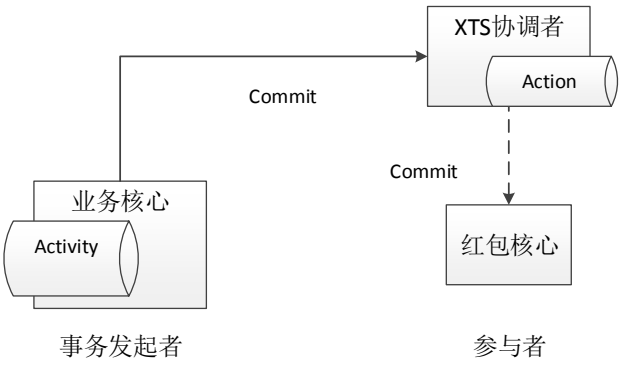
二阶段：

发起方直接向参与者发起提交/回滚操作

b. 发起方：同库模式&参与者：Remote



一阶段



二阶段

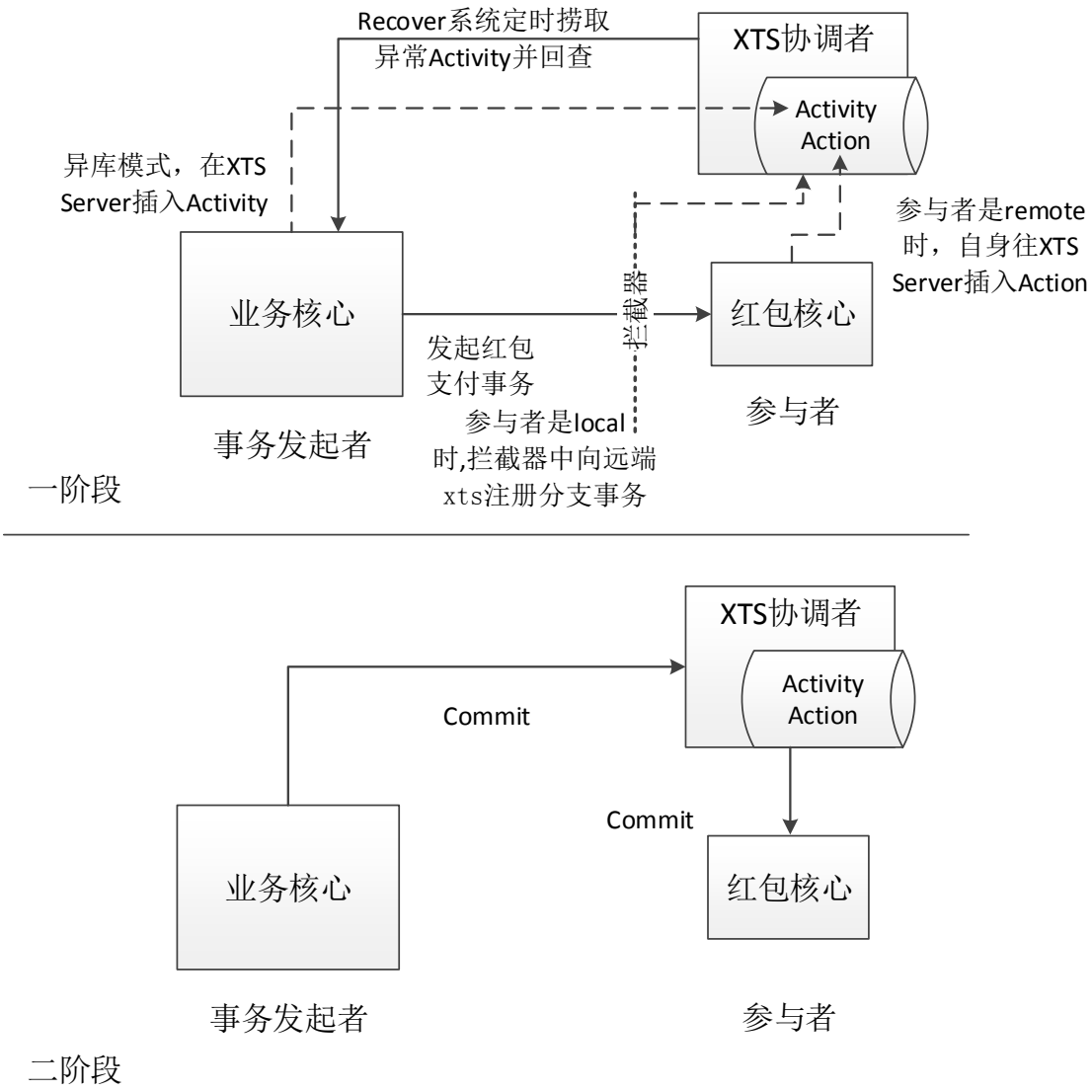
一阶段：

同库模式，Activity 记录在发起方本地，此时参与者是 remote，需要参与者自己向 XTS Server 注册分支事务

二阶段：

发起方先向 XTS Server 发送提交/回滚操作，然后 XTS Server 再向参与者发起提交/回滚操作

c. 发起方：异库模式&参与者：local/remote



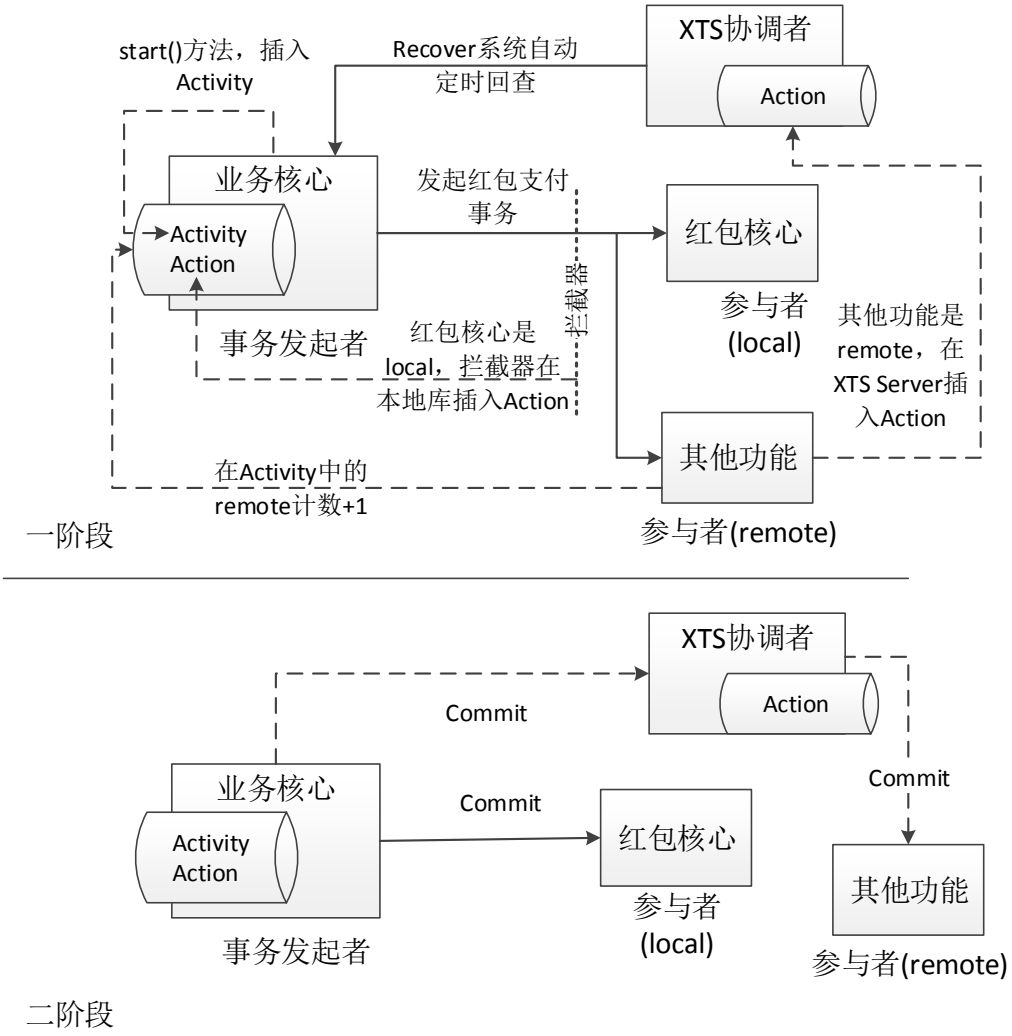
一阶段：

异库模式，Activity 记录在 XTS Server，此时如果参与者是 remote，需要参与者自己向 XTS Server 注册分支事务；如果参与者是 local，则拦截器起作用将 action 注册到 XTS Server。这里需要**注意：异库模式下，recover 是先在 XTS Server 上捞取 Activity 记录，如果二阶段提交/回滚失败，那么 recover 会调用 isDone()方法在业务方进行回查，所以需要注意一下箭头的实际含义。**

二阶段：

发起方先向 XTS Server 发送提交/回滚操作，然后 XTS Server 再向参与者发起提交/回滚操作。

d. 发起方：同库模式&参与者：mix(同时有 local 和 remote)



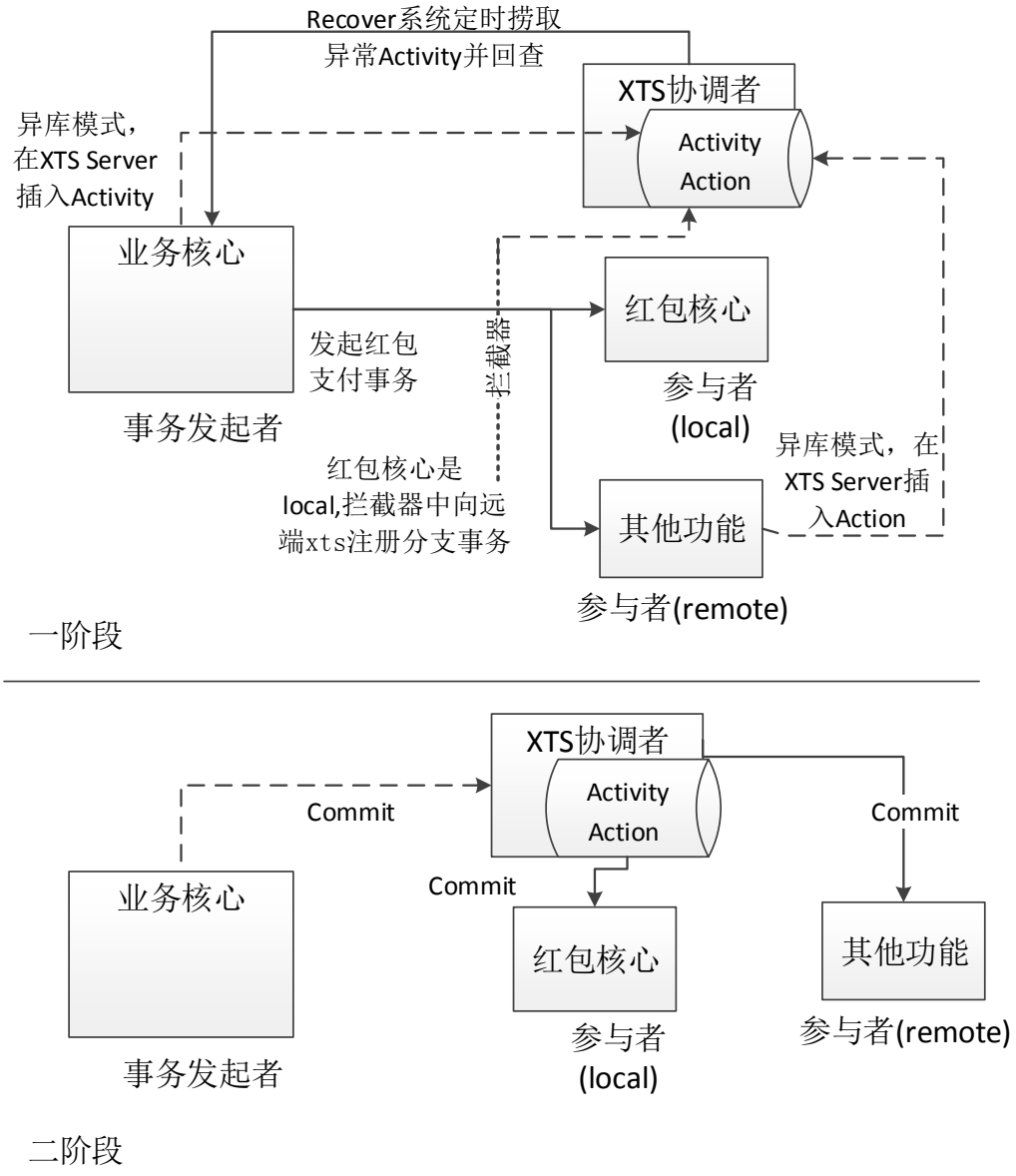
一阶段：

同库模式，Activity 记录在发起方本地库；对于 mix 模式，对 local 参与者，拦截器起作用将 action 插入本地库；对 remote 参与者，需要参与者自己向 XTS Server 注册分支事务

二阶段：

对 local 参与者，发起方直接向它提交/回滚；对 remote 参与者，发起方先向 XTS Server 发送提交/回滚操作，然后 XTS Server 再向参与者发起提交/回滚操作。

e. 发起方：异库模式&参与者：mix(同时有 local 和 remote)



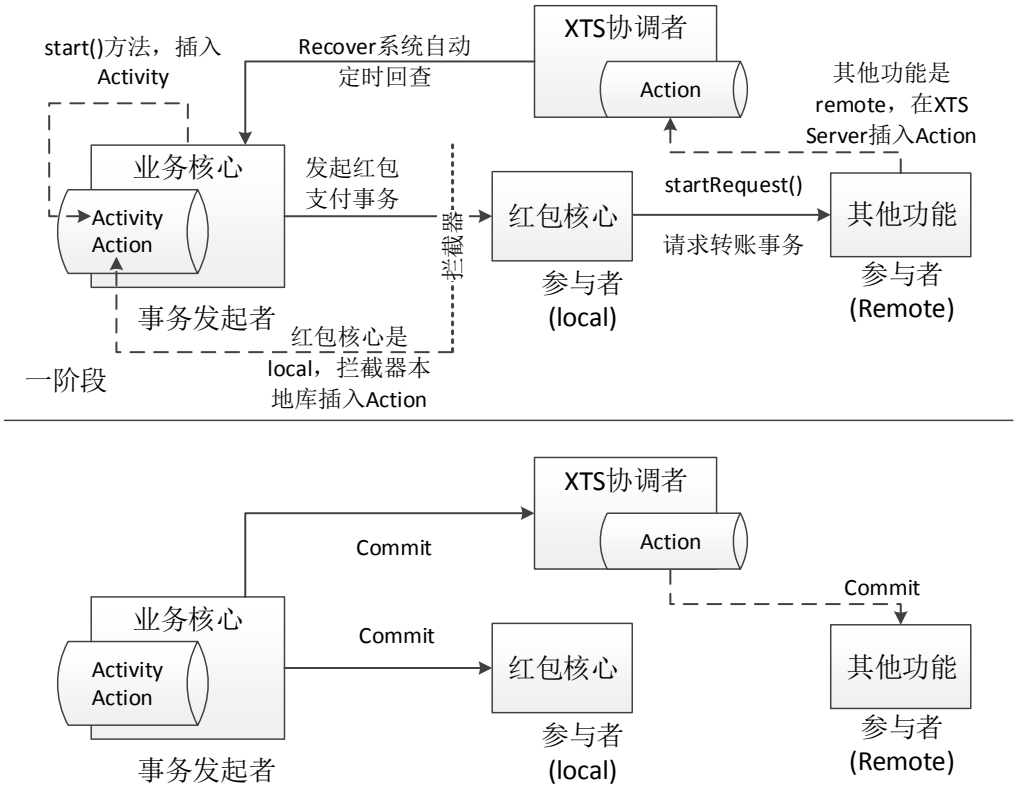
一阶段：

异库模式，Activity 记录在 XTS Server，对于 mix 模式，对 local 参与者，拦截器起作用将 action 插入 XTS Server；对 remote 参与者，需要参与者自己向 XTS Server 注册分支事务。这里需要**注意：异库模式下，recover 是先在 XTS Server 上捞取 Activity 记录，如果二阶段提交/回滚失败，那么 recover 会调用 isDone()方法在业务方进行回查，所以需要注意一下箭头的实际含义。**

二阶段：

对 local 参与者和 remote 参与者，都是发起方先向 XTS Server 发送提交/回滚操作，然后 XTS Server 再向参与者发起提交/回滚操作。

f. 发起方：同库模式&参与者：嵌套调用



一阶段：

同库模式，Activity 记录在发起方本地库；对于嵌套模式，对直接 local 参与者，拦截器起作用将 action 插入本地库；对 remote 参与者，需要参与者自己向 XTS Server 注册分支事务

二阶段：

对 local 参与者，发起方直接向它提交/回滚；对 remote 参与者，发起方先向 XTS Server 发送提交/回滚操作，然后 XTS Server 再向参与者发起提交/回滚操作。

注意：

这里，如果嵌套事务是“账务核心”，有特殊处理，实际并不会在远端 xts 注册账务分支事务，最终是否提交或回滚是由分布式事务的发起方在开启分布式事务时决定的，可以关注 BusinessActivityControlService 下的所有 start 开头的方法，里边 doStart 的时候会设置下面的枚举之一：

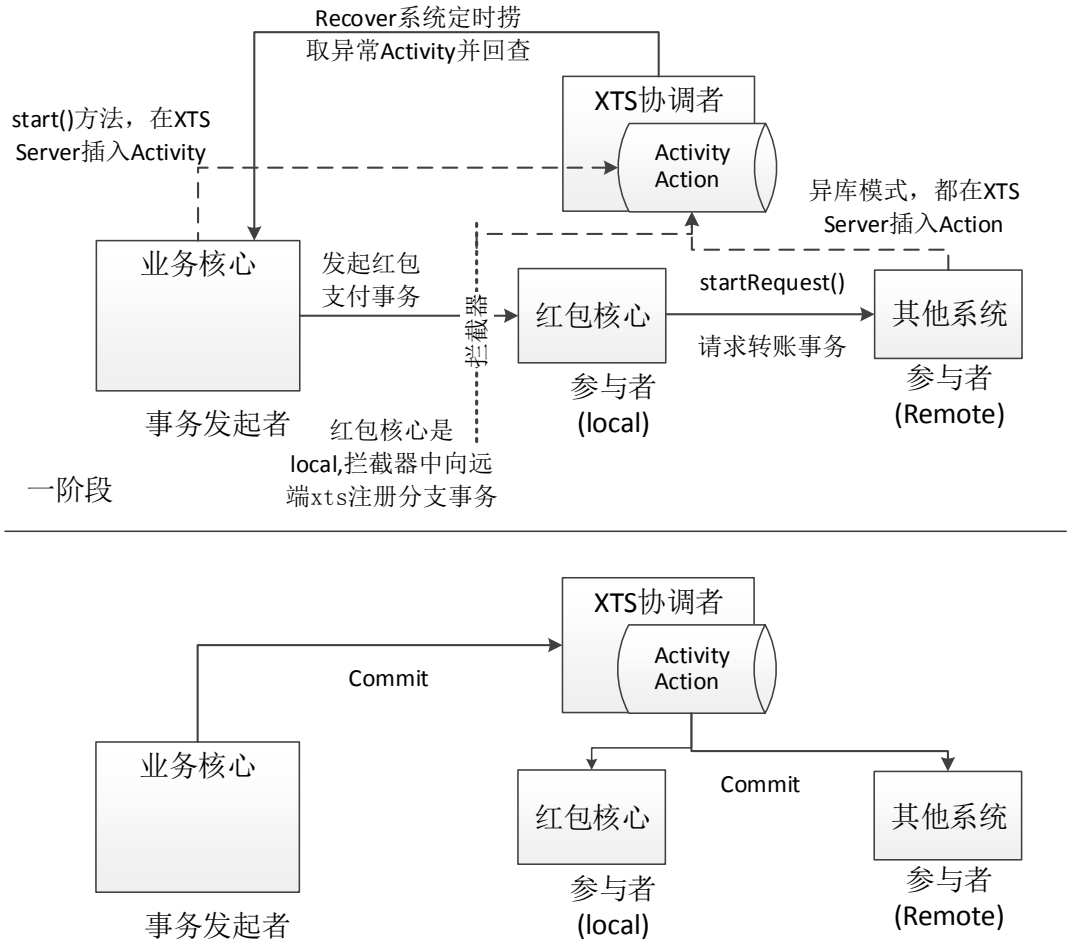
NO_TRANS('N'), //没有调用帐务

YES_ACCOUNT('Y'), //有调用帐务

YES_PAYMENT('P'); //有调用 payment

如果设置成“YES_ACCOUNT”，二阶段时会直接由发起方(同库)或 xts（异库）做账务的提交或回滚。

g. 发起方：异库模式&参与者：嵌套调用



一阶段：

异库模式，Activity 记录在 XTS Server，对于嵌套模式，对 local 参与者，拦截器起作用将 action 插入 XTS Server；对 remote 参与者，需要参与者自己向 XTS Server 注册分支事务。这里需要**注意：异库模式下，recover 是先在 XTS Server 上捞取 Activity 记录，如果二阶段提交/回滚失败，那么 recover 会调用 isDone()方法在业务方进行回查，所以需要注意一下箭头的实际含义。**

二阶段：

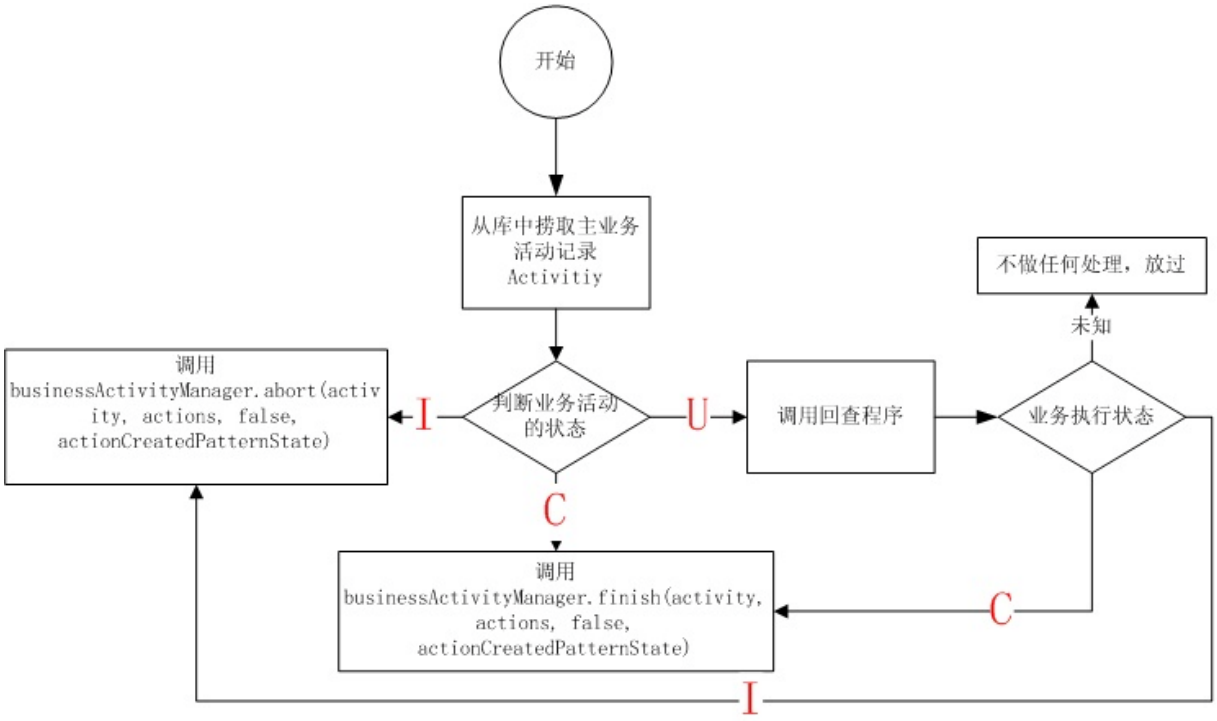
对 local 参与者和 remote 参与者，都是发起方先向 XTS Server 发送提交/回滚操作，然后 XTS Server 再向参与者发起提交/回滚操作。

以上仅对发起方和参与者一阶段 Activity 和 Action 记录的写入情况以及二阶段提交/回滚操作的流程进行总结。

2.4.3 XTS 异常处理工作原理

如果在事务执行中，一阶段失败回滚的时候，系统故障导致回滚不成功；或者二阶段提交时候，系统故障导致提交不成功，如何来保证一致性呢？XTS 系统提供了一套恢复机制来对这种出错的情况进行恢复。

XTS Server 中有一个 recover 线程一直不停地跑，每隔 1 分钟就去 Activity 记录所在的地方捞取记录，如果这是状态被记录为异常状态，那么 recover 程序就会尝试恢复被中断的事务，也就是重试，直到成功提交/回滚事务。下面这张图对这个过程已经刻画得非常清楚[6]：



可以清楚看到，如果状态为 U，表明有可能是异库模式下的主活动记录，这时不知道事务到底执行到什么程度，是否是确认提交或者确认回滚，那么需要对它进行回查，需要自己写回查程序，将回查的结果返回，如果是 DONE，表明一阶段完成，此时对应的是确认提交状态，应为 C；如果为 NOT_DONE，那么表明一阶段失败，对应了确认回滚状态，应为 I。然后再调用对应的 finish()或者 abort()方法进行提交/回滚就可以了。如果捞取出来状态为 I/C，就直接进行回滚/提交。

注意，对主活动记录捞取时是需要加锁获取的，因为怕此时还有其他操作正在进行，但是如果有这种情况：有可能当主业务记录刚刚创建成功后，存在一个很小的时间段，该记录是没有被锁住的，更巧的是恢复程序也刚好在这个时间段内捞本记录处理（因为没有加锁），这样的几率是很小的，但是确实可能发生，一旦发生后后果不堪设想：主事务还没处理完，就开始恢复了。处理方法是获取滞后的记录，现在 XTS 设置的一阶段超时时间是 30s。

2.5 小结

XTS 里的几个需要注意的地方：

- XTS 的核心思想是**两阶段提交协议**，一切操作都围绕它展开
- 发起方同库/异库模式，参与者 local/remote/mix 模式是考虑到 Activity、Action 记录存放地点，以及嵌套事务情况下的操作而产生的几种复杂的情况，本质上要理解这是为了保证一致性
- 事务的发起方可以认为是分布式事务的“协调者”，但还是称为“发起者”最贴切。要特别注意，在 XTS 中真正对应到两阶段提交中“协调者”这一个角色的应该是 XTS Server，它负责在多个参与者之间统一协调和管理，无论是多参与者还是有嵌套事务的参与者
- 事务发起方本地事务的最终状态（提交或者回滚）决定整个分布式事务的最终状态
- 分布式事务必须在本地事务中进行
- 事务参与者的方法需要支持两阶段。发起者只关注第一阶段方法，第二阶段由框架自动调用。参与者需要保证，第一阶段如果成功，第二阶段必须成功；如果不成功，则由 recover 程序不断重试来保证

Reference

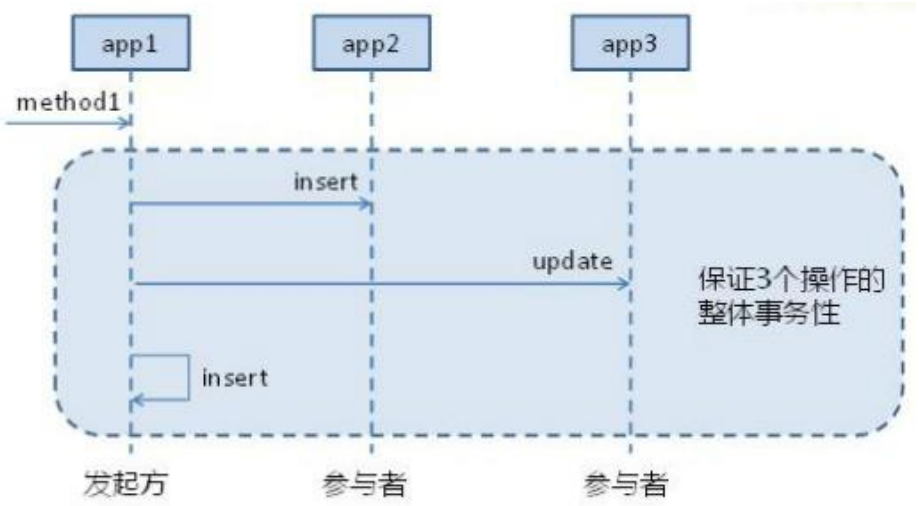
[1] 严羽. 分布式事务 XTS 原理及使用. http://atit.alipay.net/zlearning/index.php?r=courseDetail/index&courseid=594
[2] 潇桐. 数据库事务的学习笔记. http://magician8421.iteye.com/blog/2042798
[3] Artech. 谈谈分布式事务之一：SOA 需要怎样的事务控制方式.
http://www.cnblogs.com/artech/archive/2010/01/26/1657102.html
[4] 鲁肃. XTS 分布式事务-res: SOA 化金融系统的分布式事务.ppt; 支付宝分布式事务设计草案.doc.
http://doc.alipay.net/pages/viewpage.action?pageId=27202744
[5] 陈浩. 分布式系统的事务处理. http://coolshell.cn/articles/10910.html
[6] 善攻. XTS 分布式事务-开发文档-v4. http://doc.alipay.net/pages/viewpage.action?pageId=27202715

3 XTS 实例分析和配置使用

@潇桐

3.1 XTS 里的基本概念

举个例子：



这是一个调用链路，在这个调用事务中，有 app1，app2,app3 组成，APP1 需要调用 app2 的插入操作（可能是 RPC），成功后 app2 需要调用 app3 的更新操作，成功后 APP1 需要自己完成插入操作。整个事务的一致性就需要保证 app2 的 insert，app3 的 update，和 app1 的 insert 要么同时成功要么同时失败。不能说 app1 的 insert 失败了，但是 app3 的 update 确是成功的。

对比 XTS，那么 method1 称之为一个 activity，它有 app2 和 app3 两个参与者，因此 app2 的 insert 是一个 action，app3 的 update 是一个 action。

3.2 同库模式

接下来就让我们看看如何实现一个 XTS 的配置，

这里假设个场景，在一次转账过程中，包含三个步骤，web 系统调用 pay 系统完成源账户的扣钱工作，然后 web 系统调用 deposit 系统完成目标账户的存钱工作，然后 web 系统自己要记录到数据库里。

3.2.1 编写 Action

分析这个场景我们可以看到这个分布式事务由两个参与者，pay 系统和 deposit 系统，发起者是 web 系统。因此我们应该有两个 Action，payAction 和 depositAction。

因此我们需要编写这两个 payAction 和 depositAction：

PayAction 接口：

```
public interface PayAction {

    @TwoPhaseBusinessAction(name = "PayAction", commitMethod = "doCommit", rollbackMethod = "doRollback", actionType = "local")

    public boolean doPay(String txId, Long actionId);

    public boolean doCommit(String txId, long actionId);

    public boolean doRollback(String txId, long actionId);
}
```

PayAction 接口实现：

```
public class DefaultPayAction implements PayAction {
    @Override
    public boolean doPay(String txId, Long actionId) {
        System.out.println("Pay系统完成了付款的预操作");
        return true;
    }

    @Override
    public boolean doCommit(String txId, long actionId) {
        System.out.println("Pay系统提交了付款操作");
        return true;
    }

    @Override
    public boolean doRollback(String txId, long actionId) {
        System.out.println("Pay系统完成了付款操作的回滚操作");
        return true;
    }
}
```

DepositAction 接口：

```
public interface DepositAction {

    @TwoPhaseBusinessAction(name = "DepositAction", commitMethod = "doCommit", rollbackMethod = "doRollback", actionType = "local")

    public boolean doDeposit(String txId, Long actionId);

    public boolean doCommit(String txId, long actionId);

    public boolean doRollback(String txId, long actionId);
}
```

DepositAction 接口实现：

```
public class DefaultDepositAction implements DepositAction {
    @Override
    public boolean doDeposit(String txId, Long actionId) {
        System.out.println("Deposit系统完成了收款的预操作");
        return true;
    }

    @Override
    public boolean doCommit(String txId, long actionId) {
        System.out.println("Deposit系统提交了收款操作");
        return true;
    }

    @Override
    public boolean doRollback(String txId, long actionId) {
        System.out.println("Deposit系统完成了收款操作的回滚操作");
        return true;
    }
}
```

- Action 接口说明：

如前文说到，action 就是参与者的一个业务操作，而由 2PC 理论可以看到，参与者需要提供 prepare、commit、rollback 三个操作，因此 action 也需要提供三个操作，例如对于存款这个 action，需要有 doPay,doCommit,doRollback 三个操作，在 doPay 中将会把存款的基本信息先临时存放到一个地方。在 XTS 中，Action 对应的是一个类的三个方法：

PayAction 就是付款 Action 的接口，其中 doPay 就是预处理操作方法，doCommit 就是提交方法，doRollback 就是回滚方法。发起者就是调用 PayAction 来完成这个 action 操作

3.2.2 编写 Activity

Activity 是整个业务活动的第一阶段，因此在 acitivity 里我们需要包含 PayAction 的 doPay 方法，DepositAction 的 doDeposit 方法，以及自身的 doLog 方法。

因此我们的 Activity 编写如下：

```
public class TransferAccount {

    private TransactionTemplate transferTemplate = null;
    private PayAction payAction = null;
    private DepositAction depositAction = null;
    private BusinessActivityControlService businessActivityControlService = null;
    public final static String PCARD_XTS_BIZ_TYPE = BusinessActivityCatagoryUtil.ACAP
            .toString() + "001";

    public void doTransfer() {
        transferTemplate.execute((TransactionCallbackWithoutResult) (status) -> {
                //这里调用businessAcivityControlService启动分布式事务
                businessActivityControlService.start(PCARD_XTS_BIZ_TYPE, "123");
                //这里调用PayAction完成付款操作
                payAction.doPay(null, null);
                //这里调用DepositAction完成存款操作
                depositAction.doDeposit(null, null);
                //这里完成记录
                doLog();
        });
    }

    private void doLog() {
        //这里完成数据库的写入操作;
    }

    //以下是getter setter
    public TransactionTemplate getTransferTemplate() { return transferTemplate; }
```

例子红框对应的业务处理就是整个事务操作，即第一阶段。我们可以看到这个分布式事务首先是被一个叫 transferTemplate 的本地事务模板包裹，因为 doLog 操作会对本地数据库进行写入，所以 doLog 操作的回滚和提交就由 transferTemplate 控制，另外，如前所说，分布式事务的最终状态由本地事务决定，因此 transferTemplate 的提交或者回滚决定了整个事务的提交或者回滚。另外我们会看到一个 businessActivityContolService，这里先不细讲，后续章节会提到。

①本地 Mysql 数据库增加 business_activity 表和 business_action 表（Local 模式需要）

Business_activity

```
CREATE TABLE /*!32312 IF NOT EXISTS*/ "business_activity" (
  "TX_ID" varchar(20) DEFAULT NULL,
  "STATE" char(1) DEFAULT NULL,
  "ACCOUNT_TRANS_STATE" char(1) DEFAULT NULL,
  "GMT_CREATE" datetime DEFAULT NULL,
  "GMT_MODIFIED" datetime DEFAULT NULL,
  "PROPAGATION" char(1) DEFAULT NULL
);
```

Business_action

```
CREATE TABLE /*!32312 IF NOT EXISTS*/ "business_action" (
  "ACTION_ID" bigint(20) DEFAULT NULL,
  "TX_ID" varchar(40) DEFAULT NULL,
  "NAME" varchar(64) DEFAULT NULL,
  "STATE" char(1) DEFAULT NULL,
  "GMT_CREATE" datetime DEFAULT NULL,
  "GMT_MODIFIED" datetime DEFAULT NULL,
  "CONTEXT" varchar(4000) DEFAULT NULL
);
```

思考

为什么需要这两张表呢，试想，如果我们的事务在 commit 或者 rollback 时出现异常，那么需要后续重试，怎么才能让 xts 知道有哪些 activity 或者 action 呢，这个时候就需要有这两张表记录正在运行的 activity 和 action 的状态。一句话就是：local 模式需要把 action 和 activity 落地到本地，好让 XTS 在恢复中断的事务时能读取到。

②配置 spring

本地事务配置，包含一个事务管理器和一个事务模板

```
<bean id="transferTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRES_NEW"/>
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
</bean>
<!--事务管理器-->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

PayAction 的配置包含一个 payAction 的本地实现，一个被 businessActionInterceptor 代理的代理类，和一个注册到 businessActivityManager 的 bacsBusinessAction 里的扩展点。

```
<!--开始配置action-->
<bean id="payActionInLocal" class="action.impl.DefaultPayAction"/>

<bean id="payAction" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"
              value="action.PayAction"/>
    <property name="target" ref="payActionInLocal"/>
    <property name="interceptorNames">
        <list>
            <value>businessActionInterceptor</value>
        </list>
    </property>
</bean>


<sofa:extension bean="businessActivityManager"
                point="bacsBusinessAction">
    <sofa:content>
        <bacsBusinessAction>
            <serviceInterface>
                action.PayAction
            </serviceInterface>
            <bean>payAction</bean>
        </bacsBusinessAction>

    </sofa:content>
</sofa:extension>
```

DepositAction 的配置类似 PayAction，这里不做赘述

```
<bean id="depositActionInlocal" class="action.impl.DefaultDepositAction"/>

<bean id="depositAction" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"
              value="action.DepositAction"/>
    <property name="target" ref="depositActionInlocal"/>
    <property name="interceptorNames">
        <list>
            <value>businessActionInterceptor</value>
        </list>
    </property>
</bean>

<sofa:extension bean="businessActivityManager"
                point="bacsBusinessAction">
    <sofa:content>
        <bacsBusinessAction>
            <serviceInterface>
                action.DepositAction
            </serviceInterface>
            <bean>depositAction</bean>
        </bacsBusinessAction>

    </sofa:content>
</sofa:extension>
```

③同库模式自身的依赖配置

为 businessActivityManager 注册一个独立的事务模板，事务传播级别为 PROPAGATION_REQUIRES_NEW

```
<sofa:extension bean="businessActivityManager" point="bacsTransactionTemplate">
    <sofa:content>
        <bacsTransactionTemplate bean="newTransTransactionTemplate"/>
    </sofa:content>
</sofa:extension>
<!--声明事务模板-->
<bean id="newTransTransactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRES_NEW"/>
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
</bean>
```

同库模式依赖的本地库的配置(MYSQL)：

数据源配置

```
<bean id="baseSqlMapClientDAO" abstract="true">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
    <property name="sqlMapClient">
        <ref bean="sqlMapClientDAO"/>
    </property>
</bean>
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://127.0.0.1:3306/world"/>
    <property name="username" value="root"/>
    <property name="password" value="admin"/>
</bean>
<bean id="sqlMapClientDAO" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="configLocation">
        <value>sqlmap/sqlmap.xml
        </value>
    </property>
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
```

DAO 配置

```
<!--Dao扩展点-->
<sofa:extension bean="businessActivityStore" point="bacsDaoSet">
    <sofa:content>
        <bacsDaoSet>
            <businessActivityDAO>businessActivityDAO</businessActivityDAO>
            <businessActionDAO>businessActionDAO</businessActionDAO>
        </bacsDaoSet>
    </sofa:content>
</sofa:extension>
<!--dao声明-->
<bean id="businessActionDAO"
      class="com.alipay.sofa.platform.xts.bacs.store.dal.impl.IbatisBusinessActionMysqlDAO"
      parent="baseSqlMapClientDAO"/>
<bean id="businessActivityDAO"
      class="com.alipay.sofa.platform.xts.bacs.store.dal.impl.IbatisBusinessActivityDAO"
      parent="baseSqlMapClientDAO"/>
```

SqlMap 配置

```
<sqlMapConfig>
    <settings cacheModelsEnabled="true" enhancementEnabled="false" lazyLoadingEnabled="false" maxRequests="3000"
              maxSessions="3000" maxTransactions="3000" useStatementNamespaces="false"/>
    <sqlMap resource="sqlmap/common-sqlmap-mapping.xml"/>
          <sqlMap resource="sqlmap/City-sqlmap-mapping.xml"/>
       <sqlMap resource="sqlmap/BusinessAction-mysql-tddl-sqlmap-mapping.xml"/>
    <sqlMap resource="sqlmap/BusinessActivity-mysql-tddl-sqlmap-mapping.xml"/>
</sqlMapConfig>
```

其他依赖配置：

账务处理服务

```
<sofa:extension bean="businessActivityManager" point="bacsAccountTransFinalizer">
    <sofa:content>
        <bacsAccountTransFinalizer bean="accountTransFinalizer"/>
    </sofa:content>
</sofa:extension>
<!-- 账务处理完成服务，由账务核心提供  -->
<sofa:reference id="accountTransFinalizer"
                interface="com.alipay.sofa.platform.xts.bacs.spi.AccountTransFinalizer">
    <sofa:binding.ws testURL="${acctrans_service_url}">
        <compatible>
            <vip>${acctrans_service_url}</vip>
            <serviceUrl>/services/accountTransFinalizer</serviceUrl>
        </compatible>
    </sofa:binding.ws>
</sofa:reference>
```

Action 的切入类

```
<!--action的切入类-->
<bean id="businessActionInterceptor"
      class="com.alipay.sofa.platform.xts.bacs.integration.BusinessActionInterceptor">
    <property name="businessActivityControlService">
        <sofa:reference
                interface="com.alipay.sofa.platform.xts.bacs.api.BusinessActivityControlService"/>
    </property>
</bean>
```

④测试

终于配置完一堆的东西，这里我们先不说每个配置的理由，先跑一下 test 有个宏观的认识

配置一个 transferAccount：

<bean id="transferAccount" class="activity.TransferAccount"/>

并编写一个测试用例：

```
public class TestTransferXts extends JunitEclipseSofaTest {

    @XAutoWire(value = XAutoWire.BY_NAME, bundle = "xiaotong.core.service")
    protected TransferAccount transferAccount = null;


    @Test
    public void testTransferAccount() {
        transferAccount.doTransfer();
    }

}
```

如果是 IDEA，需要设置这个测试用例的 Working Directory 为 test 文件夹，然后 run，结果如下：

a. 正常提交

```
Pay系统完成了付款的预操作
09:50:55,876  INFO  BusinessActivityManagerImpl : 成功添加原子业务活动:[txid:024001-123],[actionRecord:(024001-123,8919115294479109817)[name=DepositAction,
Deposit系统完成了收款的预操作
Retrieving document at 'null'.
09:50:56,278  WARN  XFireSyncInvokeComponent :  [initXFireService,com.alipay.sofa.platform.xts.bacs.spi.AccountTransFinalizer]
09:50:56,285  WARN  XFireSyncInvokeComponent :  [initXFireProxy,com.alipay.sofa.platform.xts.bacs.spi.AccountTransFinalizer,http://10.209.100.98:8080/servi
09:50:56,517  WARN  RPCTransmitHandlerInitListener$1 : 转发dataId[SOFA_PRC_TRANSMIT_HELLOSOFA]可转发的目标地址列表为:

09:50:56,523  INFO  BusinessActivityManagerImpl : 账务提交完毕，结果为 - True,activity=024001-123(state=C,accountTransCount=2,actionCount=2,actionCountInRe
Pay系统提交了付款操作
09:50:56,525  INFO  BusinessActivityManagerImpl : 原子活动提交完毕，结果为 - True,action=(024001-123,-6690498478345469055)[name=PayAction,gmtCreate=null,gm
Deposit系统提交了收款操作
```

可以看到 Pay 和 Deposit 系统完成了预操作，然后后续完成了提交操作。查看数据库发现两张表都没有记录。

b. 显示回滚

接下来让我们看看回滚的场景，在 doTransfer()里加上 status.setRollbackOnly()表示显示回滚

```
public void doTransfer() {
    transferTemplate.execute((TransactionCallbackWithoutResult) (status) -> {
            //这里调用businessAcivityControlService启动分布式事务
            businessActivityControlService.start(PCARD_XTS_BIZ_TYPE, "123");
            //这里调用PayAction完成付款操作
            payAction.doPay(null, null);
            //这里调用DepositAction完成存款操作
            depositAction.doDeposit(null, null);
            //这里完成记录
            doLog();
            status.setRollbackOnly();
    });
}
```

运行：

```
Pay系统完成了付款的预操作
09:55:32,137  INFO  BusinessActivityManagerImpl : 成功添加原子业务活动:[txid:024001-123],[actionRecord:(024001-123,3882158986549807998)[name=DepositAction,
Deposit系统完成了收款的预操作
Retrieving document at 'null'.
09:55:32,539  WARN  XFireSyncInvokeComponent :  [initXFireService,com.alipay.sofa.platform.xts.bacs.spi.AccountTransFinalizer]
09:55:32,546  WARN  XFireSyncInvokeComponent :  [initXFireProxy,com.alipay.sofa.platform.xts.bacs.spi.AccountTransFinalizer,http://10.209.112.189:8080/serv
log4j:ERROR No output stream or file set for the appender named [_DAILY_APPENDER_NAME].
log4j:ERROR No output stream or file set for the appender named [_DAILY_APPENDER_NAME].
09:55:32,786  INFO  BusinessActivityManagerImpl : 账务回滚完毕，结果为 - True,activity=024001-123(state=I,accountTransCount=2,actionCount=2,actionCountInRe
Pay系统完成了付款操作的回滚操作
09:55:32,788  INFO  BusinessActivityManagerImpl : 原子活动回滚完毕，结果为 - True,action=(024001-123,-1854426291777550277)[name=PayAction,gmtCreate=null,gm
Deposit系统完成了收款操作的回滚操作
```

可以看到 Pay 系统和 Deposit 系统完成了回滚操作，检查数据库发现依然没有记录。

c. 设置 commit 或者 rollback 失败

接上个例子，我们把 rollback 状态设置为 false，例如把 PayAction 的 doRollback 设置为 false

```
@Override
public boolean doRollback(String txId, long actionId) {
    System.out.println("Pay系统完成了付款操作的回滚操作");
    return false;
}
```

运行：

可以看到 PayAction 的回滚操作失败了，再 check 数据库可以发现：

ACTION_ID	TX_ID	NAME	STATE	GMT_CREATE	GMT_MODIFIED	CONTEXT
-5324220654715058003	024001-123	PayAction	I	2014-07-21 10:02:21	2014-07-21 10:02:21	(NULL)
-3292999229437030697	024001-123	DepositAction	I	2014-07-21 10:02:21	2014-07-21 10:02:21	(NULL)

两个 Action 的状态都为 I，Activity 的状态也为 I，这就说明因为 PayAction 的失败，导致两个 Action 的回滚操作都失败，事务本身没有成功回滚，最后交由 XTS 轮训失败记录进行重试（XTS 系统配置了 Recovery 后由 XTS 自动调用）。

小结

- 同库模式需要本地存在 Action 和 Acivity 的数据源。
- 同库模式需要 XTS 系统配置 Recovery 才能完成异常提交的恢复。（后续总结）
- 对于成功提交或者回滚的事务，Action 和 Activity 表不会存放记录。
- 对于异常提交或者回滚的事务，Action 和 Activity 表会存在失败记录，由 Recovery 重试，因此我们需要保证 commit 和 rollback 操作的幂等性。进一步，如果在提交阶段出现错误，那么 Action 和 Acivity 记录的状态为 C，如果在回滚时出现错误，那么 Action 和 Acivity 记录的状态为 I。
- Start 方法的参数，这个 businessType 需要全局唯一，且要向 XTS 团队申请：

  BusinessType：业务活动类别

  （6 位数字）或者（5 位数字+P），最后一位为 P 的是统一支付参与的事务

  前 3 位由框架统一分配，后 3 位由业务自己定义

  BusinessId：业务活动 ID，通常具有业务含义

  txId=businessType + “-” +businessId，总长度<=128

3.3 异库模式

配置完了同库模式，你会不会有疑问，难道我每个发起者都需要自己维护一套事务的 DAO 和数据源么，而且配置这么复杂。其实 XTS 还为我们提供了公用的事务元数据存储，我们只需要通过远程调用就可以了，这就是异库模式。因此，异库模式和同库模式的区别就在于 Activity 和 Action 的存储是和业务数据库在一个库还是不同库。如果在一起就是同库模式（需要是同一个数据库实例，在一个物理机而不在同一个实例

都不算同库模式），如果不在一起（Action 和 Activity 存放在 XTS 中）那么就为异库模式。

因为 Action 和 Activity 在 XTS 系统里，因此所有的启动、提交和回滚操作需要委托 XTS 来做，因此异库模式的配置需要同时更改发起者，还要更改 XTS 系统。所以说 XTS 是一个比较重的系统，因此本次示例并不方便更改 XTS 系统（需要伴随项目开 XTS 分支），这里只是根据现成例子提几点。

①首先是发起者，因为发起者不需要再存放 activity 和 action 信息，因此就不需要配置那么一套 DAO 和 DS，只要 BusinessActivityManager 没有设置 newTransactionTemplate 字段，XTS 就认为你采用了异库模式

```
<!--    <sofa:extension bean="businessActivityManager" point="bacsTransactionTemplate">
        <sofa:content>
            <bacsTransactionTemplate bean="newTransTransactionTemplate"/>
        </sofa:content>
    </sofa:extension>-->
```

a. 这里注销掉 bacsTransactionTemplate 的扩展点。

b. 然后配置 XTS 的远程服务

```
<sofa:reference id="businessActivityRemote"
                interface="com.alipay.sofa.platform.xts.bacs.facade.BusinessActivityRemoteManagerFacade">
    <sofa:binding.ws testURL="${xts_service_url}">
        <compatible>
            <vip>${xts_service_url}</vip>
            <serviceUrl>/services/businessActivityRemoteManagerFacade</serviceUrl>
        </compatible>
    </sofa:binding.ws>
</sofa:reference>
<sofa:extension bean="businessActivityManager" point="remoteManagerService">
    <sofa:content>
        <remoteManagerService timeout='30'>
            <bean>
                businessActivityRemote
            </bean>
        </remoteManagerService>
    </sofa:content>
</sofa:extension>
```

这个 BusinessActivityRemote 就是 XTS 暴露的用于提交和回滚的远程服务，同时也提供服务供发起者将 action 和 activity 存放到远端。

c. 接下来配置一个状态回查的接口：

回查接口供 XTS 的 recovery 程序使用。因为在异库模式下，action/activity 表并不受本地事务控制，例如在同库模式里，如果本地事务异常中断，action 和 activity 的状态变更并不会受本地事务的最终状态进行回滚，所以无法保证 ACID，因此 recovery 程序在执行重试时是无法知道事务的最终状态的，必须通过发起者的回查接口判断。另外，就算第一阶段成功结束，事务可以提交或者回滚，事务状态也为 U，这是为啥呢，个人认为还是 Action/Activity 表不受本地事务控制也无法加锁，因此对于 XTS recovery 事务的状态仍然是未知的，所以不论什么时候都是 U。

1) 接着配置参与者，需要把 action 的暴露成 sofa 服务（这里不做举例，参考 sofa 的 service 配置）

2) 配置 XTS 端，这里需要 checkout 一个 xts 的分支，然后配置参与者的 action，同时需要配置 recovery，整个配置过程较为类似配置同库模式。

```
<sofa:extension bean="businessActivityManager" point="bacsBusinessAction">
    <sofa:content>
        <bacsBusinessAction>
            <bean>
                DailyCutService
            </bean>
            <serviceInterface>
                com.alipay.innertrans.facade.api.DailyCutSupportFacade
            </serviceInterface>
        </bacsBusinessAction>
    </sofa:content>
</sofa:extension>
<sofa:extension bean="businessActivityManager" point="bacsBusinessAction">
    <sofa:content>
        <bacsBusinessAction>
            <bean>
                PcardcoreService
            </bean>
            <serviceInterface>
                com.alipay.pcardcore.common.service.facade.PcardFundFacade
            </serviceInterface>
        </bacsBusinessAction>
    </sofa:content>
</sofa:extension>
```

配置 recovery 的细节会在后文里详细阐述。

3.4 理解同库和异库模式

先看看官方 PPT 里的总结：

同库 vs 异库 （针对发起方）

	优点	缺点	适用场景	备注
同库模式	1. 对分布式事务活动表的操作都在业务方本地进行，不需要额外的远程调用，速度快。 2. 对xts系统无依赖（无remote参与情况） 3. 业务方使用简单，不需要写回查	1. 分布式事务活动表在业务库内，管理不便，出现问题后不易排查 2. Xts恢复系统需要为同库模式配置数据源	业务量较大的系统	当业务方的分布式事务会影响到至少5%的交易量时，建议使用同库模式。
异库模式	1. 所有的分布式事务活动都集中管理，管理方便，易于排查问题	1. 速度较同库模式慢 2. 对xts系统有依赖 3. 业务方需要写回查	业务量较小的系统	当业务方的分布式事务不会对交易量有较大影响时，建议使用异库模式。

细分来说，主要有 remote 模式，local 模式，和 MIX 模式。

REMOTE 模式：没有配置本地的 ds，action 和 activity 存储在 XTS，整个回滚和提交过程将委托 XTS 进行。

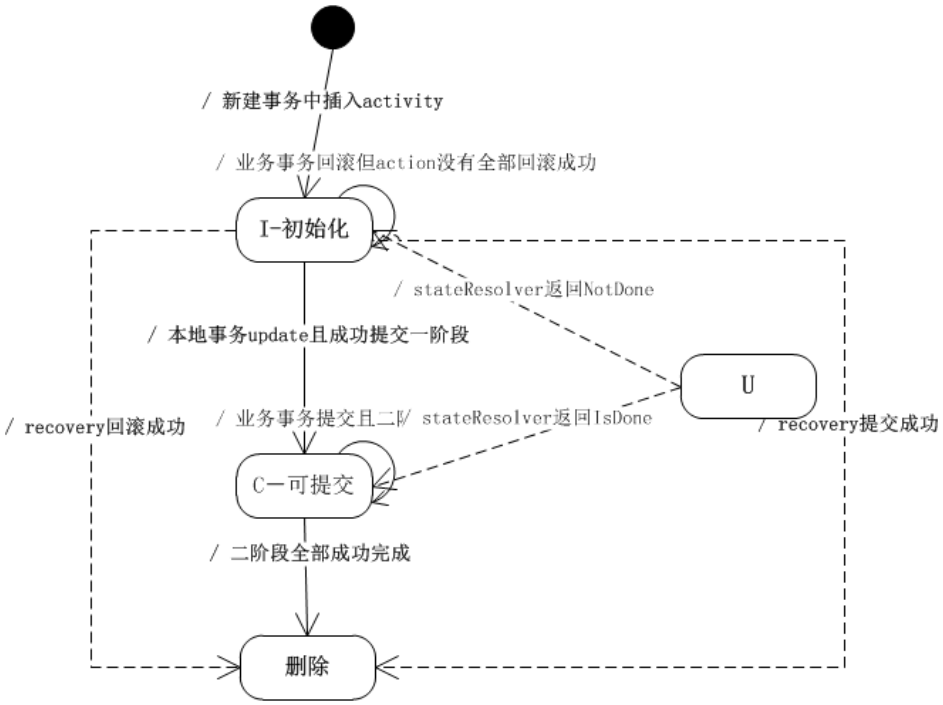
LOCAL 模式：事务元数据存放在本地，且没有 remote 的直接参与者，那么发起者只会调用本地 action 进行提交和回滚。

MIX 模式：即事务元数据存放在本地，但是有 remote 参与者，那么发起者会先调用 XTS 提交、回滚 remote 参与者，成功后再调用本地 action 进行提交和回滚。

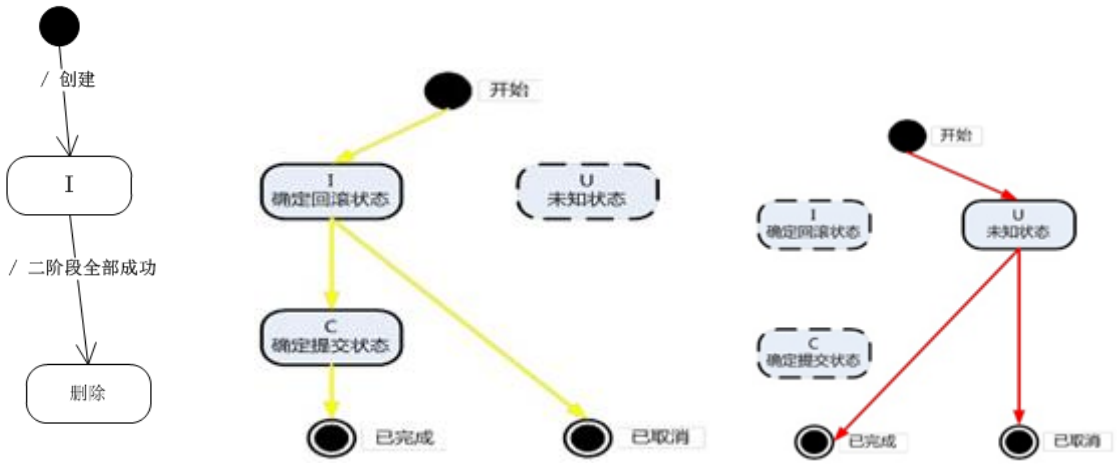
```
public void finish(BusinessActivityRecord activity, List<BusinessActionRecord> actions,
                   ActionCreatedPatternState actionCreatedPatternState) {

    if (actionCreatedPatternState == ActionCreatedPatternState.Remote) {
        businessActivityRemoteManagerFacade.doCommit(activity.getTxId().toStringForm(), true,
            activity.getState().getCode());

    } else if (actionCreatedPatternState == ActionCreatedPatternState.Local) {
        doFinishLocalActionAndAccount(activity, actions);

    } else if (actionCreatedPatternState == ActionCreatedPatternState.Mix) {
        //先处理远程原子活动，处理正确后再处理本地
        if (businessActivityRemoteManagerFacade.doCommit(activity.getTxId().toStringForm(),
            false, activity.getState().getCode())) {
            doFinishLocalActionAndAccount(activity, actions);
        } else {
            logger.warn("业务活动: " + activity.getTxId().toStringForm() + "远程提交原子活动失败");
        }
    } else {
        throw new BusinessActivityIllegalConfigException("出现了不应该的错误ActionCreatedPatternState: "
                                                         + actionCreatedPatternState);
    }
}
```

已经总结的非常详细，这里笔者就不再赘述，需要提到的就是不同模式下状态的变更：

Activity 在同库模式下的状态变化：



上图为 Action 在同库模式下的状态变化：



左图为 Activity 在异库模式下的状态变化：U→删除；中图为同库模式 activity 状态变化简单版；右图为异库模式状态变化简单版

3.5 参与者的 local/remote 模式

Local 模式和 remote 模式是相对于参与者来说的，即参与者(action 是存放在本地还是存放在 XTS 中)，如果 action 在 prepare 操作里调用 enrollAction 将自己添加到 XTS 里，那么这个 Action 就是 remote 模式。跟发起者无关，即发起者可以是同库，也可以是异库。

我们需要在申明 Action 的注解里注明是 remote 模式：

例如：

```
@TwoPhaseBusinessAction(name = "DepositAction", commitMethod = "doCommit", rollbackMethod = "doRollback", actionType = "remote")
```

Remote 模式下 XTS 框架做了什么？

- 正常情况下，调用 Action 的 prepare 方法时，会将 Action 的记录写入 Action 表，而如果 xts 框架检测到这个 ActionType 是 remote，那么他就不会创建 Action 记录。而只是记录 remote 数。
- 因此，每个 remote 的参与者需要在自己的 prepare 方法中，显示调用 enrollAction 方法将自己添加到 XTS 系统的 Action 表里。
- 同时 XTS 根据 remote 数来判断发起者是本地模式还是 MIX 模式

```
    } else {
        /*
         * 该模式的actiontype对应于remote，原子活动存储在远程端
         */
        activity.increaseActionCountToRemote();
    }
```

如果是 mix 模式，那么发起端不仅仅要提交本地 Action 还要提交 remote action。

3.6 嵌套事务

分布式环境下的嵌套事务是指，参与者里又调用了下一个分布式参与者，而这个参与者也需要纳入分

布式事务的管理。

嵌套事务在整个事务调用链路里使用的是发起者启动的 txid，不会主动建立 activity。按照 xts 的规范，嵌套事务的提交和回滚者是发起者本身，提交和回滚事务只能处理本地事务（线程上下文中）的事务和 remote 的事务(XTS)，因此必须要求嵌套事务的 action 在 XTS 中，这样事务发起者在委托 XTS 提交或者回滚 action 的时候，才能被顺利完成。而 remote 模式则是天生的适合嵌套事务的模型，嵌套参与者只需要设置为 remote 模式，然后在 XTS 中配置 sofa 服务，同时直接调用者使用 startRequires 就可以完成将嵌套事务的处理。

事务参与者在调用 remote 的参与者时，需要使用 startRequires 方法。这个方法不会创建 activity 记录，而只是复用父 activity，同时给 activity 标记为 REQUIRE 模式，在提交时，如果遇到 Require 模式则不会做真正的提交操作，由最初的事务发起者提交事务。因此，对于有嵌套事务的场景，事务的最初发起者必须是 MIX 或者 REMOTE 模式（这样才能委托 XTS 提交 remoteaction）

PS：由上可知，其实只要 xts 中具有 action 的记录就能完成事务提交或者回滚，所以这样也是可以实现嵌套事务的：

发起者（异库模式）→参与者(businessActivityManager 异库模式)→参与者(local)

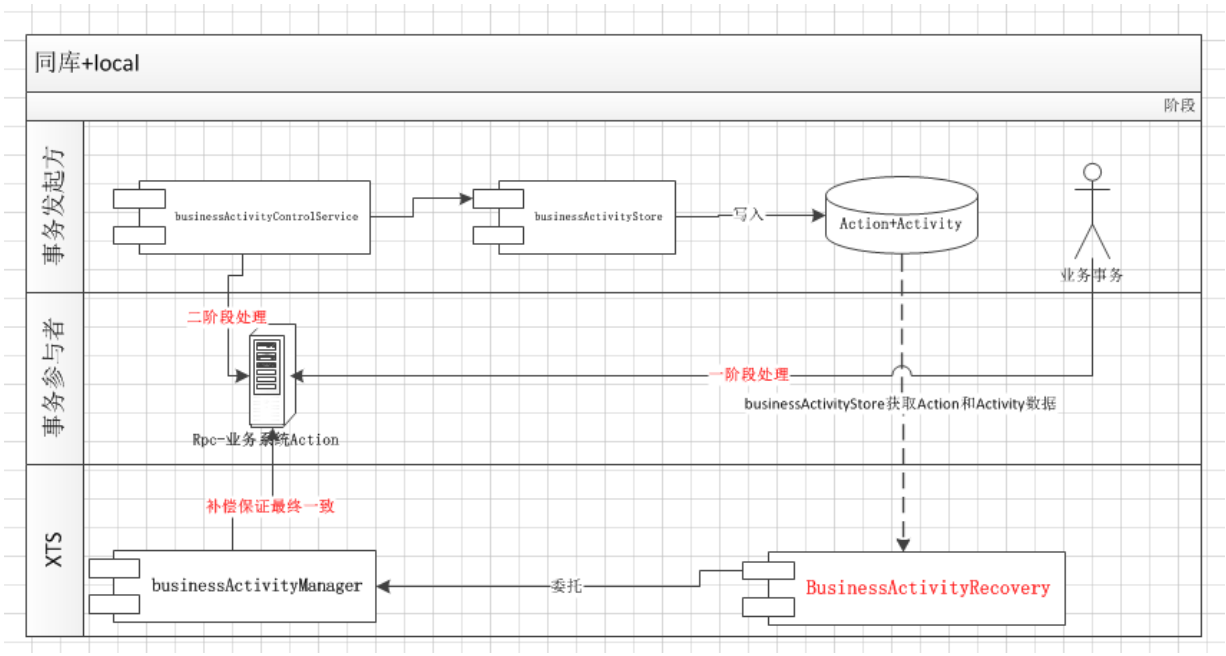
3.7 模式小结

不同发起者下 local 和 remote 模式：

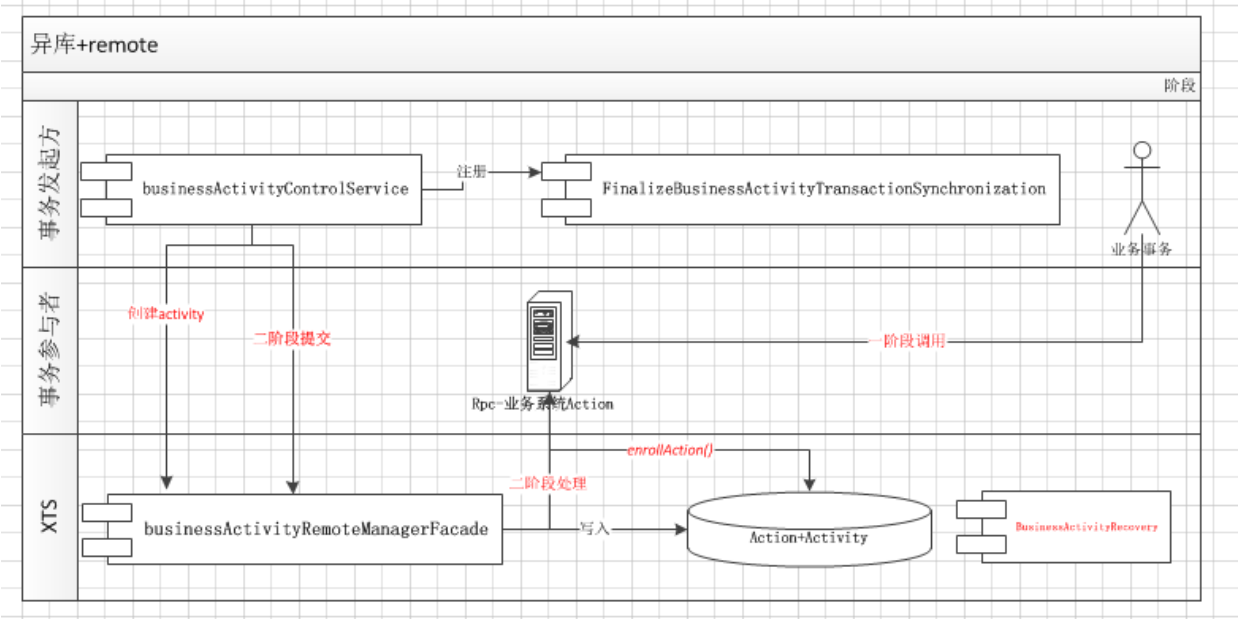
发起者	参与者	执行情况
同库	local	单纯的同库模式，发起者作为事务的协调者
异库	local	因为发起者使用了异库模式，整个调用会被转化为单纯的异库模式
同库	remote	即 MIX 模式，会先提交 remote，再提交本地
异库	remote	即单纯的异库模式

图解模式：

同库+local



异库+remote(local)



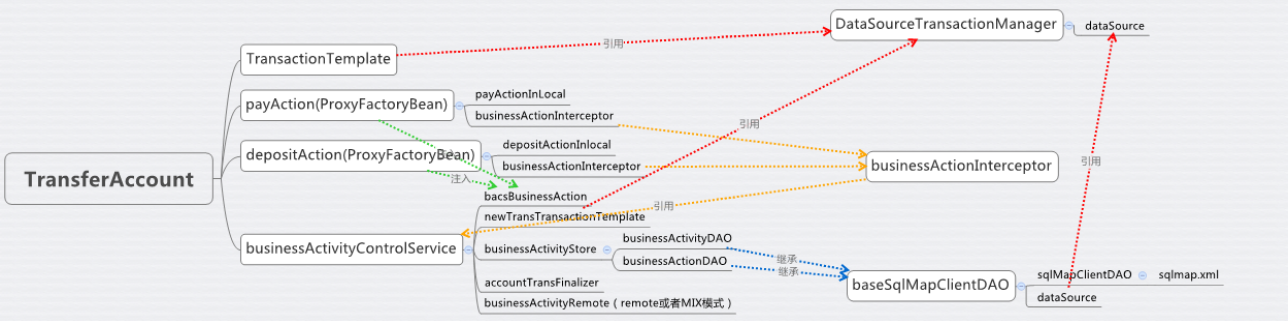
Mix 模式就是发起者是同库模式同时含有 remote 的参与者，

因此第一阶段发起者需要调用本地配置的 Action 和 remote 的 Action，但只有本地 Action 的记录会被写入到本地库。

第二阶段不仅要提交 remoteaction 还要提交本地 action。对于 RemoteAction 则委托 XTS 的 BusinessActivityRemoteManagerFacade 进行提交或者回滚。

3.8 详解配置（同库模式）

XTS 的配置非常多，新同学可能会很奇怪为啥要那么配置，这里对 XTS 的配置做简要说明，方便大家理解。



这张图描述了 TransferAccount 这个类的配置之间的依赖关系：

组件	说明
TransactionTemplate	本地事务，决定整个分布式事务的启动或者回滚
PayAction/DepositAction	其实是利用 AOP 实现的代理类，包装了 PayAction 和 DepositAction 的真实实现，其实在同库模式里写入 Action 就是在代理类里完成的。
BusinessActivityControlService	分布式事务的启动核心类， ● 关联一个 bascBusinessAction，用于注册所有的 Action。 ● 一个 newTransactionTemplate，这里是为了让 action 和 activity 的插入不受外部事务影响。 ● 一个 businessAcitivityStore 用于存储本地的 Action 和 Activity ● 一个 accountTransFinalizer，账务的特殊处理 ● BusinessActivityRemote：对于异库模式需要依赖的服务

思考：

XTS 发起者如何保证即使本地事务的提交和回滚不影响 activity 表的插入和更新？即为什么要新建 newTransactionTemplate

答案：

是为了让 Activity 和 action 的 Insert 操作不受外部事务影响，始终能插入成功。XTS 对 activity 和 action 表的操作使用了独立的事务模板，事务传播机制为 new，即独立于本地事务，重新开启一个新事物，本地事务的提交和回滚不影响 activity 表的 DAL，同时 activity 的 DAL 的回滚依然会影响到主事务。

如果 insert 操作在单独的事务模板里，无论本地事务是否提交或者回滚，acitivivity 的记录都存在，但是如果插入失败，独立事务模板在回滚后将异常抛给外部事务导致本地事务也回滚，实现一致性。而为什么 update 方法会放在外面，是因为如果本地事务失败，就需要把 update 状态给回退到初始化状态，同时如果 update 失败只需要回滚状态而不用删除记录。保证 activity 表里始终存在事务记录有理可循。

补充：事务的传播特性

事务的传播特性主要是针对嵌套事务的，

例如有

```
 A.method(){
  startTransaction();
  B.mehod();
  endTransaction();
}
B.method(){
  startTransaction();
 //dosomething;
  endTransction();
}
```

对于 A 嵌套 B 事务的情况，如果希望 B 事务的回滚和提交不受 A 事务的影响，即 A 回滚的 B 也不回滚，那么 B 事务的传播特性应该为 PROPAGATION_REQUIRES_NEW，相反应该为 PROPAGATION_REQUIRED。

为什么会有这种场景，例如在付款事务里，包含记日志的事务，当我们希望不论付款是否成功，日志系统都有记录，那就应该用 requires-new 事务。

另外 spring 如何实现的呢？

其实就是当事务模板在获取一个事务时，当发现当前模板使用的是 REQUIRED，那么他会复用已经存在的事务，当他发现时 NEW 的时候，那么就会重新创建连接，重新创建事务，让两个事务之间不相互影响。(一个 DB 连接无法开启两个事务)

4　XTS 主要流程源码浅析

@柳成

本章主要对发起方/参与者的主要执行流程源码进行一个探索，看看 XTS 是如何实现之前所说的二阶段提交协议并确保一致性的。由于 XTS 代码还是比较多，很多细节处理不会展示，主要还是将关键流程中的关键代码贴出来并进行分析。关于国际账务支付部分没有进行分析，而且不同版本的源码在结构上可能有些调整，这里展示的源码的版本是较新的 xts core-4.4.10。

4.1 发起方流程（一阶段）

通过调用下面方法开始发起一次分布式事务：

```
businessActivityControlService.start(CardConstants.PCARD_XTS_CARD_BIND, pcardNo);
```

然后会进入 BusinessActivityControlService.doStart()方法，开启事务，并注册事务同步器：

BusinessActivityControlServiceImpl.java

```
private void doStart(BusinessActivityId txId, BusinessActivityPropagation propagation,
                     BusinessTransState transState, String userId) {
    // 业务活动必须在本地事务中启动，理由分析见P31页“思考”
    if (!TransactionSynchronizationManager.isActualTransactionActive()) {
        throw new BusinessActivityIllegalStateException(null, "不存在本地事务，无法启动业
务活动");
    }
    try {
        BusinessActivityRecord activity = businessActivityManager.start(txId,
propagation, transState, userId);
        BusinessActivityContextHolder.setCurrent(activity);
    } catch (RuntimeException e) {
        BusinessActivityContextHolder.clear();
        throw e;
    } finally {
        if (BusinessActivityContextHolder.isActive()) {
            // 注册事务同步器，在事务提交或回滚后结束业务活动
            TransactionSynchronizationManager.registerSynchronization(new
FinalizeBusinessActivityTransactionSynchronization(
                    businessActivityManager));
        }
    }
}
```

其中 businessActivityManager.start()方法就是实际开启分布式事务，它的动作主要是新增 Activity 记录，会调用到 addInitBusinessActivity()方法：

DefaultBusinessActivityManagerImpl.java

```
private BusinessActivityRecord addInitBusinessActivity(final BusinessActivityId txId,
BusinessActivityPropagation propagation, BusinessTransState transState, final String
userId) {
    ……
    //对于REQUIRES类型的事务，如果已经存在同样的事务上下文，需要继承原有的事务上下文 不进行新事务的创建动作。
    if (propagation == BusinessActivityPropagation.REQUIRES) {
        if (logger.isInfoEnabled()) {
            logger.info("启动的事务将加入外围分布式事务业务活动，事务号[" + txId + "]" + ",
事务传播属性为[REQUIRES]");
        }
        //嵌套事务不需要去查询activity，在分库情况下他不知道在哪个库里
        BusinessActivityRecord parentActivity = new BusinessActivityRecord(txId);
        //为了处理payment还款嵌套分布式事务的问题，不允许随意修改
        parentActivity.setAccountTransCount(1);
        parentActivity.setActionCount(1);
        parentActivity.setPropagation(propagation); // 设置业务活动传播属性
        //线程变量存储活动状态
        businessActivityType.setType(BusinessActivityCreatedPatternState.NEST);
        BusinessActivityContextHolder.setCurrentActivityType(businessActivityType);
        return parentActivity;
    }
    final BusinessActivityRecord activity = BusinessActivityRecordFactory
            .createActivityRecord(txId);
    activity.setTransState(transState);
    activity.setUserId(userId);
    activity.setPropagation(propagation); // 设置业务活动传播属性
    initInsubclass(activity); // 子类特殊初始化
    //判断是否走远程服务调用
    if (activityRecordedInRemote) {
        // @author jiajia.lijj 调用远程服务持久化activity记录
        businessActivityRemoteManagerFacade.addBusinessActivity(txId.toStringForm(),
activity.getTransState().getCode(), activity.getRequestContext().currentContext());
        //远程服务的执行与当前的请求不同库，状态为UNKNOWN
        activity.setState(BusinessActivityState.UNKNOWN);
        businessActivityType.setType(BusinessActivityCreatedPatternState.ASY);
        BusinessActivityContextHolder.setCurrentActivityType(businessActivityType);
        //双重保险，确保发起方使用的是同库模式
    } else {
```

```
        boolean isSameRM = isSameRM();
        if (isSameRM) {
            activity.setState(BusinessActivityState.INIT);
        } else {
            processIsNotSameRM(activity);
        }
        // 二阶段异步化判断,二阶段异步化只支持同库模式 如果是二阶段异步化，则将主事务记录插入另外一张表。 并且不做本次的二阶段提交。
        boolean asyn = isAsync(txId);
        // 标记本次事务是否做二阶段异步化
        activity.setActivityAsync(asyn);
        // 在新事务中增加一条新的业务活动记录
        newTransactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                if (activity.isActivityAsync()) {
                    businessActivityStore.addBusinessActivityAsync(activity);
                } else {
                    businessActivityStore.addBusinessActivity(activity);
                }
            }
        });
        // 当主活动的数据库与业务活动管理器的数据库是同一个数据库时，更新业务活动状态为待提交。
        // 这样做的目的有两个:
        // 1.锁住该活动记录，防止其它业务并发修改，特别是恢复程序检查业务活动状态时并发修改。
        // 2. 将业务活动状态更新为C，但不提交事务，而是与主活动所在的业务活动中统一提交事务。
        // TODO: 如果更新结果不为1，也应该出错终止
        if (isSameRM) {
            if (activity.isActivityAsync()) {
                businessActivityStore.updateBusinessActivityAsyncState(txId,
                        BusinessActivityState.COMMIT);
            } else {
                businessActivityStore.updateBusinessActivityState(txId,
                        BusinessActivityState.COMMIT);
            }
            businessActivityType.setType(BusinessActivityCreatedPatternState.SYN);
        }
        BusinessActivityContextHolder.setCurrentActivityType(businessActivityType);
    }
    return activity;
}
```

这段代码有点长，但是比较重要。可以看到，先对事务传播模型进行判断，如果是 REQUIRES 属性，表明是一个嵌套事务，那么将由发起嵌套事务的参与者执行业务活动，而不用再新增 Activity 记录；如果是

REQUIRES_NEW 属性，说明是发起方发起一个新的分布式事务，需要新增 Activity 记录。

然后根据 activityRecordedInRemote 判断是否是异库模式，如果是，那么将会 RPC 调用远程服务 businessActivityRemoteManagerFacade.addBusinessActivity()在 XTS Server 上进行 Activity 注册；如果是同库模式，那么需要再次去实际判断一下主活动数据库是否和与业务活动管理器同库，再次确定之后再将 Activity 记录插入到本地库中（忽略代码中“二阶段异步提交”，那是提高并发量的措施，这里不做讨论）。

最后立刻将活动记录状态改为 COMMIT，目的在源码注释中写得很清楚，利用 update 加锁的特性，如果一阶段还没执行完，recover 就来捞取 Activity 记录进行重试，就会发生错误，对记录加锁可以防止 recover 程序错误的并发修改，就算后续操作出了问题，还可以将状态改为 I。

回到 doStart()方法的代码中来看，在完成开启动作之后，必须马上去注册事务同步器：TransactionSynchronizationManager.registerSynchronization()。事务同步器的作用是在完成一阶段准备工作后，框架会自动调用事务同步器中的 beforeCommit(), beforeCompletion(), afterCompletion()：

FinalizeBusinessActivityTransactionSynchronization.java

```
//该方法只有在事务提交前没有发生异常或者没有被显示调用 status.setRollbackOnly()才会被spring
事务框架触发
@Override
public void beforeCommit(boolean readOnly) {
    ……
    //判断本地事务的提交是否超时，对于嵌套的分布式事务 和 同库模式不去判断
    if (activity.getGmtCreate() != null &&
businessActivityManager.isActivityRecordedInRemote()) {
        if (activity.getGmtCreate().compareTo(
            new Date(System.currentTimeMillis() - businessActivityManager.getTimeOut() *
1000)) < 0) {
            throw new RuntimeException("分布式事务的运行时间超过了设定的有效时间:"
                                       + businessActivityManager.getTimeOut() + "s");
        }
    }
}

//该方法不管是事务发起是否发生异常，在事务最后提交的时候都会触发
@Override
public void beforeCompletion() {
    ……
    BusinessActivityPropagation propagation = activity.getPropagation();
    //只有在REQUIRES_NEW模式下获取数据才有意义
    if (propagation == BusinessActivityPropagation.REQUIRES_NEW) {
        actionCreatedPatternState = ActionCreatedPolicyGenerator.getPolicy(activity
            .getActionCountInRemote(),
businessActivityManager.isActivityRecordedInRemote());
```

```
        actions = activity.getActionRecordsInLocal();
        ……
    }
}

//在本地事务结束后调用，执行业务活动的实际提交或回滚。
@Override
public void afterCompletion(int status) {
    ……
    if (status == STATUS_COMMITTED && activity.getState() == BusinessActivityState.INIT)
{
        activity.setState(BusinessActivityState.COMMIT);
    }
    ……
    if (status == STATUS_COMMITTED) {
        if (propagation == BusinessActivityPropagation.REQUIRES) {
            ……
            try {
                businessActivityManager.finish(activity, actions,
actionCreatedPatternState);
            } catch (Exception e) {
                logger.error("无法提交业务活动[" + currentId + "]，留待系统自动恢复", e);
            }
        } else {
            ……
        }
    } else if (status == STATUS_ROLLED_BACK) {
        if (propagation == BusinessActivityPropagation.REQUIRES) {
            ……
        } else if (propagation == BusinessActivityPropagation.REQUIRES_NEW) {
            ……
            try {
                businessActivityManager.abort(activity, actions,
actionCreatedPatternState);
            } catch (Exception e) {
                logger.error("无法回滚业务活动[" + currentId + "]，留待系统自动恢复", e);
            }
        } else {
           ……
        }
    } else {
        logger.warn("由于事务结束状态[" + status + "]未知，不进行处理，留待系统自动恢复");
    }
```

```
}
```

其中 beforeCommit() 和 beforeCompletion() 的区别是一阶段没有发生异常便会调用 beforeCommit() 方法，而不管有没有异常都会调用 beforeCompletion() 方法。beforeCommit() 方法的作用主要是对 remote 模式下判断本地事务是否超时；beforeCompletion()方法主要是获得 action 的 State 判断是 local，remote 或者 mix；在 afterCompletion()方法中就会对 activity 进行提交/回滚，其中 finish()方法会根据 local/remote/mix 模式分别调用 doCommit(), doFinishLocalActionAndAccountOrPayment() 和 doFinishRemoteLocalAndAccountOrPayment()方法来进行提交，然后发起分布式事务这块流程就算完结。

4.2 参与者流程（一阶段）

对于参与者来说，首先是需要配置拦截器的：（以 pcardcore 为例）

pcardcore-common-service-integration-xts.xml

```
<!-- 原子业务调用事务控制注射器 -->
<bean id="businessActionInterceptor"
     class="com.alipay.sofa.platform.xts.bacs.integration.BusinessActionInterceptor" />
```

pcardcore-common-service-integration-minitrans.xml

```
<!-- 预付卡账务AOP代理 -->
<bean id="pcardMinitransFacade"
class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"
value="com.alipay.minitrans.service.facade.pcard.api.PCardTransFacade" />
    <property name="target" ref="pcardMinitransFacadeTarget" />
    <property name="interceptorNames">
        <list>
            <value>businessActionInterceptor</value>
        </list>
    </property>
</bean>
```

那么再调用参与者 PCardTransFacade 中的方法的时候，拦截器会拦截该调用并且进入 BusinessActionInterceptor 类并优先执行一些操作。

BusinessActionInterceptor 中对于不同模式的参与者会进行判断：

BusinessActionInterceptor.java

```
    // local 模式处理
if (actionType.equals(XTSConstant.LOCAL)) {
    //设定 acitonID
```

```
    doActionInLocalType(invocation, actionName, txId);
} else if (actionType.equals(XTSConstant.REMOTE)) {
   //该模式的 actiontype 对应于 remote，原子活动存储在远程端【其插入库中的行为由业务方自己实现】
    activity.increaseActionCountToRemote();
} else if (actionType.equals(XTSConstant.INTER)) {
    //纯粹给国际支付定制
    doActionInInterType(invocation, actionName, txId);
} else {
    throw new BusinessActivityIllegalConfigException("用户配置的业务类型" + actionType + "
框架无法感知!");
}
```

① 如果对于 local 模式，那么 action 的提交由框架完成，具体的代码在 doActionInLocalType()方法中：

BusinessActionInterceptor.java

```
/**
 * local 模式原子活动的处理<p>
 * <li>第二个参数必须是 actionid
 * @param invocation       调用方法对象
 * @param actionName       原子活动名称
 * @param txId            txid
 */
private void doActionInLocalType(MethodInvocation invocation, String actionName,
                                 BusinessActivityId txId) {
    if (invocation.getArguments()[1] != null) {
        throw new BusinessActivityException("原子业务处理请求中的 actionId 不允许人工指定。
[txid:" + txId.toStringForm() + "][actionName:" + actionName + "]");
    }
    Properties context = fetchActionRequestContext(invocation);
    // 将原子业务活动加入到业务活动中
    BusinessActionRecord actionRecord =
businessActivityControlService.enrollAction(txId.toString(), actionName, context);
    invocation.getArguments()[1] = actionRecord.getActionId();
}
```

这里 businessActivityControlService.enrollaction()会将 action 记录添加到表中：

BusinessActivityControlServiceImpl.java

```
// 在新事务中增加一条新的业务原子活动
businessActionRecord = (BusinessActionRecord) newTransactionTemplate
        .execute(new TransactionCallback() {
            public Object doInTransaction(TransactionStatus status) {
                return businessActivityStore.addBusinessAction(txId, actionName,
                        context, BusinessActionState.INIT);
            }
        });
```

```
if (businessActionRecord != null) {
    activity.addActionRecoredToLocal(businessActionRecord);
    activity.increaseActionCount();
    activity.increaseActionCountToLocal();
    if (logger.isInfoEnabled()) {
        logger.info("成功添加原子业务活动:[txid:" + activity.getTxId().toString() + "]"
                + ",[actionRecord:" + businessActionRecord + "]");
    }
}
```

其中 businessActivityStore.addBusinessAction()就是添加到数据源中，之后再对 Activity 业务活动的信息进行更新，插入该 Action 的信息。

② 如果是 remote 模式，原子活动 action 将存储在远程端，进入拦截器之后，这里只会将 activity 中 remote 模式 action 计数增加 1：

```
activity.increaseActionCountToRemote();
```

```
public void increaseActionCountToRemote() {
     this.actionCountInRemote++;
 }
```

因此对于 remote 模式的参与者，实际上拦截器没有做太多事情的。然后便会去接着调用参与者。那么在参与者那边，接着就会自己调用 enrollAction()方法，将 action 记录到 XTS Server 上去。那么具体做的事情是什么呢？ businessActivityControlServer.enrollAction() 方法实际上会调用 BusinessActivityManager.enrollAction() 方法，其中向 XTS Server 提交 action 方式的如下：

DefaultBusinessActivityManagerImpl.java

```
//当前请求处于异库模式下才认为是 remote 模式添加原子活动
if (activityRecordedInRemote) {
    ……
    BusinessActivityRecord activityRecord = BusinessActivityContextHolder.current();
    Properties requestContext = null;
    if (activityRecord != null && activityRecord.getRequestContext() != null) {
        requestContext = activityRecord.getRequestContext().currentContext();
    }
    //调用远程服务持久化原子业务活动
    businessActionRecord = BusinessActivityRecordFactory
            .restoreActionRecord(businessActivityRemoteManagerFacade.addBusinessAction(
                    txId.toStringForm(), actionName, context, requestContext));
    if (businessActionRecord == null) {
        throw new BusinessActivityException(txId, "【R】原子业务活动添加失败[txid:" +
txId.toString() + "]" + "[actionName" + actionName + "]");
    }
} else {……}
```

其中 businessActivityRemoteManagerFacade.addBusinessAction()就是 RPC 调用远端的方法进行 action 记录的添加。

关于这段代码有两个需要注意的地方：

a. 如果发起方是同库模式 ，而参与者是 remote 模式，当执行到上面这段代码时，乍一看发起方模式的判断条件 activityRecordedInRemote 为 false，代表了非异库模式，也就是同库模式，那么就不会执行这段代码来将 action 记录到 XTS Server 上去，而是记录到本地。这样当然是不对的。

实际上，发起方是不会跑这段代码的，这段代码实际上是在 remote 参与者这边执行。关于 activityRecordedInRemote 这个变量，实际上是这样的：如果为同库模式，那么在发起方这边是需要去配置数据源以及 DAO 的，配置了之后，DAO 以扩展点的形式注册进来，这时会调用一个 registerTransactionTemplate 来表明已经注册过了，这里会设置 activityRecordedInRemote = false，因此发起方这个变量其实是为 false，也就是同库模式。

BusinessActivityManagerImplSofa.java

```
private void registerTransactionTemplate(BacsTransactionTemplateDescriptor contribution) {
    //扩展，表明是采用旧版的方式进行 activity 记录存储
    activityRecordedInRemote = false;
    setNewTransactionTemplate(contribution.getTransactionTemplate());
}
```

那么当拦截器执行完动作之后，就会进入到参与者的执行流程中，发起方和参与者是两个系统，activityRecordedInRemote 初始化的时候是设置为 true 的，因此会进入异库模式下的流程，执行 businessActivityRemoteManagerFacade.addBusinessAction()进行远程调用，将 action 插入到 XTS Server。

画一个图可能更好理解：

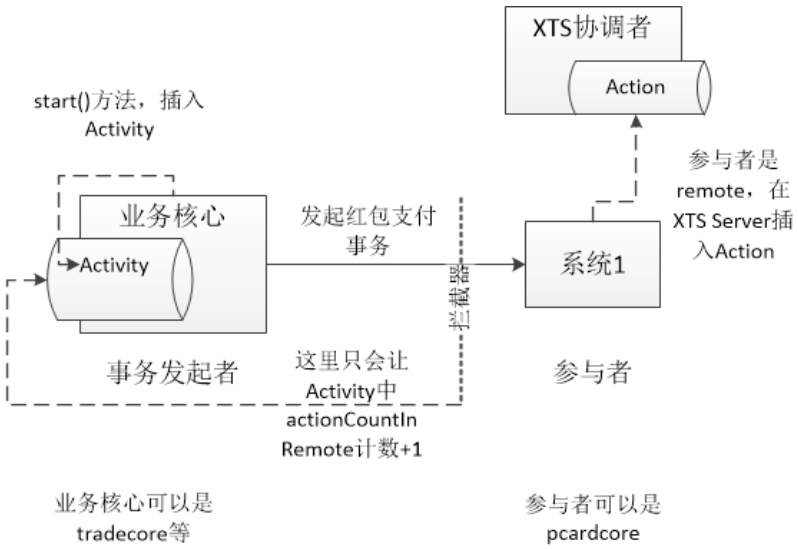


图 4.1 异库模式下 remote 参与者插入 action 记录流程

b. 注意注释“//当前请求处于异库模式下才认为是 remote 模式添加原子活动”。它代表的含义是：如果发起方是异库模式，而参与者是 local 模式，这是拦截器进入并执行到这里，需要把参与者当成 remote 模式来处理，因为异库模式下参与者的 action 记录必须放到 XTS Server 中，因此在异库模式下，local 的

参与者实际上是被当成 remote 来对待的。

4.3 参与者提交/回滚过程（二阶段）

当发起方快要完成的时候，那么框架会自动调用事务同步器中 beforeCompletion()方法，它的作用是为了获得 action 的模式：

FinalizeBusinessActivityTransactionSynchronization.java

```
if (propagation == BusinessActivityPropagation.REQUIRES_NEW) {
    actionCreatedPatternState = ActionCreatedPolicyGenerator.getPolicy(activity
        .getActionCountInRemote(), businessActivityManager.isActivityRecordedInRemote());
    actions = activity.getActionRecordsInLocal();
    if (logger.isDebugEnabled()) {
        logger.debug("在提交事务前，获取本地原子活动 actions=" + actions.size());
    }
}
```

其中 ActionCreatedPolicyGenerator.getPolicy()的具体实现为：

ActionCreatedPolicyGenerator.java

```
public static ActionCreatedPatternState getPolicy(int remoteActionCount,
                                                  boolean isActivityRecordInRemote) {
    //业务活动主记录在远程，模式为 remote
    if (isActivityRecordInRemote) {
        return ActionCreatedPatternState.Remote;
    }
    //业务活动在本地，含有 remote 参与者，模式为 mix
    if (remoteActionCount > 0) {
        return ActionCreatedPatternState.Mix;
    } else {
        return ActionCreatedPatternState.Local;
    }
}
```

这里可以非常清楚地看到，在 XTS 的框架模型下，如果发起方是异库模式，那么对于 Requires_New 的参与者，都是当成 remote 模式来对待的；而如果在同库模式下，如果有 1 个以上的 remote 参与者，那么这种状态定义为 mix 模式；如果一个 remote 参与者都没有，这时候才对待为 local 模式。

然后当发起方完成分布式事务的时候，事务同步器执行 afterCompletion()方法，在事务传播模式为 Requires_New 的时候，会触发 finish()方法：

DefaultBusinessActivityManagerImpl.java

```
public void finish(BusinessActivityRecord activity, List<BusinessActionRecord> actions,
ActionCreatedPatternState actionCreatedPatternState) {
    if (actionCreatedPatternState == ActionCreatedPatternState.Remote) {
```

```
        businessActivityRemoteManagerFacade.doCommit(activity.getTxId().toStringForm(),
true, activity.getState().getCode(), activity.getRequestContext() == null ? null:
activity.getRequestContext().currentContext());
    } else if (actionCreatedPatternState == ActionCreatedPatternState.Local) {
        doFinishLocalActionAndAccountOrPayment(activity, actions);
    } else if (actionCreatedPatternState == ActionCreatedPatternState.Mix) {
        doFinishRemoteLocalAndAccountOrPayment(activity, actions);
    } else {
        throw new BusinessActivityIllegalConfigException("出现了不应该的错误
ActionCreatedPatternState: " + actionCreatedPatternState);
    }
}
```

这里可以很清楚地看到，如果是 remote 模式，那么就会进行 RPC 调用远程接口进行提交，也就是发起方向 XTS Server 的接口进行调用，让 XTS Server 向参与者发起提交的操作，businessActivityRemoteManagerFacade.doCommit()就是 RPC 调用接口；如果是 local 模式，就会在本地执行提交操作；如果是 mix 模式，那么会先完成 remote 参与者的提交，再完成 local 参与者的提交操作。上面所有“提交”操作都对应着参与者中声明的 commitAction()方法。

分三种情况来看：

① 参与者是 local 模式

local 模式下，这段代码会在发起方执行。

调用过程是 doFinishLocalActionAndAccountOrPayment()->doFinishLocalActions()，其中会将 action 从 list 中取出，然后对每一个 action 进行提交：

DefaultBusinessActivityManagerImpl.java

```
for (BusinessActionRecord action : actions) {
    BusinessActionDescriptor actionDescriptor = businessActionRegistry
            .getBusinessAction(action.getName());
    if (actionDescriptor == null) {
        ..................
    }
    //对于 actionGroup 的方法只请求一次
    if (StringUtils.isBlank(actionDescriptor.getActionGroup())
            || actionGroup.add(actionDescriptor.getActionGroup())) {
        if (!commitBusinessAction(actionDescriptor, action,
            activity.getRequestContext())) {
            result = false;
        }
    }
}
```

当 doFinishLocalActions()执行完成之后，doFinishLocalActionAndAccountOrPayment()会开启线程池将对应的 action 记录删掉：

DefaultBusinessActivityManagerImpl.java

```
BusinessActivityRecordReaper reaper = new BusinessActivityRecordReaper(
        businessActivityStore, activity, actions, newTransactionTemplate,
        TddlHelper.current());
businessActivityReaperExecutor.execute(reaper);
```

② 参与者是 remote 模式

remote 模式下，发起方调用 businessActivityRemoteManagerFacade.doCommit()方法，对应的 doCommit 代码会在 XTS Server 上执行：

BusinessActivityRemoteManagerFacadeImpl.java

```
public boolean doCommit(final String txId, boolean isActivityRecordedInRemote,
                        final char activityCurrentState,
                        final Properties requestContext) {
    ……
    ……
    //如果主记录是在远程端，发起方进行活动提交
    if (isActivityRecordedInRemote) {
        // 使用 "for update nowait"，避免由于一条业务活动记录处于处理过程中造成恢复线程被阻塞
        // 事务提交
        Object record = transactionTemplateForRemote.execute((new TransactionCallback() {
            BusinessActivityRecord currentActivity  = null;
            //在分布式库中处理原子活动的结果
            boolean                doneDistribution = false;
             public Object doInTransaction(TransactionStatus status) {
                ……
                ……
                //原子业务活动处理
                doneDistribution = commitActions(actionsOfRemote,
                    currentActivity.getRequestContext());//和 activity 一起删除
                /**
                 * 账务处理
                 */
                boolean doneAccount = doTransCommit(currentActivity);
                /*
                 * 如果 activity 记录在分布式库中，则进行删除
                 * 在删除前确保所有的原子活动、账务提交成功
                 * 进行，异步删除（防止拥塞）
                 */
                if (doneDistribution && doneAccount) {
                    BusinessActivityRecordReaper reaper = new BusinessActivityRecordReaper(
                        businessActivityStoreForRemote, txID, actionsOfRemote,
```

```
                    transactionTemplateForRemote);
                businessActivityReaperExecutor.execute(reaper);
                if (XTSREMOTELOGGER.isInfoEnabled()) {
                    XTSREMOTELOGGER.info("分布式事务提交成功, txId=" +
currentActivity.getTxId());
                }
            } else {
                XTSREMOTELOGGER.error("分布式事务提交失败, txid=" +
currentActivity.getTxId());
            }
            ......
            ......
            return doneDistribution && doneAccount;
        }
    }));
    if (null != record) {
        return (Boolean) record;
    } else {
        return false;
    }
} else
//进行远程原子业务活动的处理
{
    //其他情况
    //原子业务活动处理
    boolean doneDistribution = commitActions(actionsOfRemote, new
XtsRequestAppContext(
        requestContext));
    //删除原子活动，同步删除，因为activity记录是在本地，需要确保原子业务活动都执行完毕后其
才能继续执行
    if (doneDistribution) {
        doneDistribution = deleteActionByTxIdAndActionId(txID, actionsOfRemote);
    }
    return doneDistribution;
}
}
```

很长的代码，但是这里要注意的有一点，isActivityRecordedInRemote 是分支判断条件，用来判断发起方是同库还是异库模式并执行不同的代码。这里 isActivityRecordedInRemote 是作为参数传进来的， 来看一下这里 doCommit()方法是如何被调用的：

```
if (actionCreatedPatternState == ActionCreatedPatternState.Remote) {
    businessActivityRemoteManagerFacade.doCommit(activity.getTxId().toStringForm(),
true, activity.getState().getCode(), activity.getRequestContext() == null ? null
                : activity.getRequestContext().currentContext());
}
```

可以发现 isActivityRecordedInRemote 传参时赋值为 true，为什么是 true？因为此时发起方为异库模式，这时无论参与者是 remote 还是 local，都会吧 action 记录在 XTS Server 上，那么统一视为 remote 模式。因此进入到相应的分支下执行代码，主要是执行每个参与者 commitAction()方法，完成之后开启线程池清除 action 记录以及 activity 记录。

那 isActivityRecordedInRemote 传参时赋值为 false 的情况呢？这种情况主要是在参与者为 mix 模式下实现的，下面会紧跟着介绍。

③ 参与者是 mix 模式

这时会先让 XTS Server 执行 commit 操作，再让发起方本地执行代码：

DefaultBusinessActivityManagerImpl.java

```
private boolean doFinishRemoteLocalAndAccountOrPayment(final BusinessActivityRecord
activity, final List<BusinessActionRecord> actions) {
    // 是否所有原子活动与 trans 处理都已全部提交
    boolean done = true;
    if (!doFinishAccountOrPayment(activity)) {
        done = false;
    }
    if (!doFinishRemoteActions(activity)) {
        done = false;
    }
    if (!doFinishLocalActions(activity, actions)) {
        done = false;
    }
    if (done) {
        BusinessActivityRecordReaper reaper = new BusinessActivityRecordReaper(
                businessActivityStore, activity, actions, newTransactionTemplate,
                TddlHelper.current());
        businessActivityReaperExecutor.execute(reaper);
        if (logger.isInfoEnabled())
            logger.info("业务活动: " + activity + "提交成功 !success");
    } else {
        logger.error("业务活动: " + activity + "提交失败! fail");
    }
    return done;
}
```

可以看到是先执行提交 remote 参与者，再执行提交 local 参与者，最后开启线程池来对所有的 action 记录进行删除。在 doFinishRemoteAction()方法中会调用：

```
businessActivityRemoteManagerFacade.doCommit(activity.getTxId().toStringForm(), false,
activity.getState().getCode(), activity.getRequestContext() == null ? null :
activity.getRequestContext().currentContext())
```

可以发现 isActivityRecordedInRemote 传参时赋值为 false。为何为 false？mix 模式的含义是，发起方

是同库，而 1 个以上参与者是 remote。那么此时，需要先进行 remote 提交，再将本地记录的 action 提交。因此，先调用 RPC 方法将 XTS Server 上记录的原子活动提交，然后删掉 XTS Server 上的活动记录，完成之后，再提交本地的原子活动并删除相应的本地记录。此时就进入了上面所说的另一个分支：

```
if (isActivityRecordedInRemote) {
    ……
    ……
}else
    //进行远程原子业务活动的处理
    {
        //其他情况
        //原子业务活动处理
        boolean doneDistribution = commitActions(actionsOfRemote, new
XtsRequestAppContext(
            requestContext));
        //删除原子活动，同步删除，因为activity记录是在本地，需要确保原子业务活动都执行完毕后其才能继续执行
        if (doneDistribution) {
            doneDistribution = deleteActionByTxIdAndActionId(txID, actionsOfRemote);
        }
        return doneDistribution;
    }
```

这里需要注意的点是：在提交完成后，删除记录的操作不是开启多线程进行的，而是同步处理，原因是由于 Activity 记录在发起方，而这段代码却在 XTS Server 上跑，需要确保将操作的结果正确返回，如果成功，那么再继续执行 doFinishLocalActions()方法，提交本地的操作，最后再删除本地的 action 记录。当然如果没有 local 的参与者，这两步就实际上没有动作。

以上是参与者在二阶段 COMMIT 操作的相关流程，而对于 ROLLBACK 操作的流程和上面的是一样的，只是关注的点是 FinalizeBusinessActivityTransactionSynchronization.afterCompletion()方法中 abort()发方法，跟踪代码发现它的流程和提交是几乎一样的，这里不再重复贴一遍源码。

4.4 XTS 恢复机制执行流程

在 XTS 的概念中，如果一阶段出了问题无法提交，那么就会对一阶段进行 rollback，不执行二阶段；如果一阶段成功，进入二阶段提交过程，这时二阶段出了问题无法提交成功，XTS 是不会对二阶段进行 rollback 的，XTS Server 的 recover 系统会自动捞取没有提交成功的参与者进行重试直到成功提交。

recover 系统主要在两种情况下产生作用：1) 一阶段成功，但是提交时出现问题，导致二阶段提交失败；2) 一阶段失败，但是在回滚时出现问题，导致回滚失败。XTS 中的 recover 系统主要就是为了恢复这些因出错而中断的事务，保证整个事务的一致性。

recover 会在 XTS 服务器上不断运行，设置间隔时间为 1 分钟，每分钟会对 Activity 记录进行捞取，将

出错的事务进行恢复。在 BusinessActivityRecoverService 中：

BusinessActivityRecoverServiceImpl.java

```
public int recover(String dbIdentify) {
.................
// 从指定的DB里捞数据
TddlHelper.setCurrent(dbIdentify);
List<BusinessActivityRecord> activities = new ArrayList<BusinessActivityRecord>();
if (recoverRollbackRecordOnly) {
    activities =
businessActivityStore.findPendingBusinessActivitiesForRollbackOnly(reconverLimit);
} else {
    activities = businessActivityStore.findPendingBusinessActivities(reconverLimit);
}
......
......
RecoveryStat stat = new RecoveryStat(0);
for (BusinessActivityRecord activity : activities) {
    // 异步恢复
    BusinessActivityRecoveryImpl businessActivityRecovery = new
BusinessActivityRecoveryImpl(
        activity);
    businessActivityRecovery.setBusinessActivityManager(businessActivityManager);
    businessActivityRecovery.setBusinessActivityStore(businessActivityStore);
    businessActivityRecovery.setStateResolvers(stateResolvers);
    businessActivityRecovery.setDbIdentify(dbIdentify);
    businessActivityRecovery.setNewTransactionTemplate(newTransactionTemplate);
    businessActivityRecovery.setRecoveryMode(recoveryMode);
    xtsRecoverExecutor.execute(businessActivityRecovery);
}
.................
}
```

businessActivityStore.findPendingBusinessActivities()就是对数据库中的记录进行捞取，然后开启多线程对每个 Activity 进行恢复处理：(rocover 这个方法名称是一种什么节奏？)

BusinessActivityRecoveryImpl.java

```
public RecoveryStat rocover(final BusinessActivityRecord activity) {
    final RecoveryStat stat = new RecoveryStat(1);

    if (activity.getState() == BusinessActivityState.UNKNOWN) {
        try {
            doResolveUnknown(activity, stat);
```

```
        } catch (Exception e) {
            logger.error("解析未知业务活动[" + activity + "]状态时出现异常", e);
        }
    }
    if (activity.getState() == BusinessActivityState.COMMIT
        || activity.getState() == BusinessActivityState.INIT) {
        try {
            doSingleRecover(activity, stat);
        } catch (Exception e) {
            logger.error("恢复业务活动[" + activity + "]时出现异常", e);
        }
    }
    ……
    ……
    return stat;
}
```

对于 UNKNOWN 状态的事务，recover 程序必须先进行回查，回查的接口由系统实现时在 stateResolver 中给出。先看 doResolveUnknown()的逻辑：

BusinessActivityRecoveryImpl.java

```
private void doResolveUnknown(BusinessActivityRecord activity, RecoveryStat stat) {
    ……
    BusinessActivityStateResolver stateResolver = stateResolvers.get(businessCategory);
    if (stateResolver == null) {
        ……
        return;
    }
    int isDone = stateResolver.isDone(businessType, businessId);

    // 可以由stateResolver来保证，比如通过锁定记录的形式。
    if (isDone == BusinessActivityStateResolver.DONE) {
        stat.incTotalUnknownToCommit();
        businessActivityStore.updateBusinessActivityState(txId,
BusinessActivityState.COMMIT);
        activity.setState(BusinessActivityState.COMMIT);
    } else if (isDone == BusinessActivityStateResolver.NOT_DONE) {
        stat.incTotalUnknownToRollback();
        businessActivityStore.updateBusinessActivityState(txId,
BusinessActivityState.INIT);
        activity.setState(BusinessActivityState.INIT);
    } else {
      ……
```

```
      ……
    }
      ……
}
```

其中 isDone()就是回查接口，需要在对应的 stateResolver()中写相应的回查逻辑，回查的返回结果是两种：DONE 以及 NOT_DONE。如果是 DONE，表明一阶段已经成功，需要进入确定提交状态 C；如果是 NOT_DONE，表明一阶段失败，需要回滚，进入确定回滚状态 I。那么接下来就会将对应的 activity 状态转换为 BusinessActivityState.COMMIT 或者 BusinessActivityState.INIT。

接下来进入到具体的恢复流程 doSingleRecover()：

BusinessActivityRecoveryImpl.java

```
private void doSingleRecover(final BusinessActivityRecord activity, final RecoveryStat
stat) {//进行数据加锁获取
    if (ActionCreatedPatternState.getByCode(recoveryMode) !=
ActionCreatedPatternState.Remote) {
        newTransactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                //加锁获取，防止捞取的数据正在处理中
                BusinessActivityRecord currentActivity = null;
                try {// 使用 "for update nowait"，避免由于一条业务活动记录处于处理过程中造成
恢复线程被阻塞
                    currentActivity = businessActivityStore
                        .getBusinessActivityForUpdate(activity.getTxId());
                } catch (CannotAcquireLockException e) {
                    ……
                }
                if (currentActivity == null) {
                    // 业务活动已不存在，可能正在处理中，也可能已经处理完成，不进行恢复
                    return;
                }
                // 检查业务活动状态是否有改变，若有改变，表明有其它程序还在处理中，为避免竞争冲
突，此时不进行恢复
                if (activity.getState() != currentActivity.getState()) {
                    logger.warn("【R】业务活动[" + currentActivity + "]状态发生改变，新业务
活动为[" + currentActivity + "]，暂不恢复");
                    return;
                }
                recover(currentActivity, stat);
            }
        });
    } else {
```

```
            recover(activity, stat);
        }
    }
```

这个地方要注意的是，首先这里判断了在不为 remote，也就是 local/mix 情况下，首先对 Activity 记录进行加锁获取，然后判断记录中的状态是否和当前状态一致，为何要这样做呢？由于是 local/mix，说明 Activity 记录在发起方，这时有可能当本地事务结束时，锁被释放，但可能存在一个很小的时间段，恢复程序也刚好在这个时间段内捞本记录处理（因为没有加锁），这样的几率是很小的，但是确实可能发生，一旦发生后后果不堪设想：主事务还没处理完，就开始恢复了。处理方法是获取滞后的记录，现在 XTS 设置的一阶段超时时间是 30s，因此这里是将记录的状态和当前状态作比较，如果不一致则说明还在处理，暂不恢复。

如果一切符合条件，则开始恢复：

BusinessActivityRecoveryImpl.java

```
private void recover(BusinessActivityRecord activity, RecoveryStat stat) {
    ……
    ……
    boolean result = false;
    /**
     * 在恢复端，其捞取数据就相当于在本地处理，所以出于本地的情况就是mix和local
     * 1：对于发起方异库的，恢复阶段对应为LOCAL
     * 2：对于发起方是同库的，但是含有remote模式的参与者，恢复阶段对应为MIX
     * 3：对于发起方是同库的，全部的原子活动都是local模式的，恢复阶段对应为LOCAL
     */
    if (state == BusinessActivityState.COMMIT) {
        try {
            businessActivityManager.finish(activity, actions, ActionCreatedPatternState
                .getByCode(recoveryMode));
            stat.incTotalCommitSuccess();
            result = true;
        } catch (Exception e) {
            logger.error("提交业务活动[" + txId + "]时出现异常", e);
            stat.incTotalCommitFail();
        }
    } else if (state == BusinessActivityState.INIT) {
        try {
            businessActivityManager.abort(activity, actions, ActionCreatedPatternState
                .getByCode(recoveryMode));
            stat.incTotalRollbackSuccess();
            result = true;
        } catch (Exception e) {
            logger.error("回滚业务活动[" + txId + "]时出现异常", e);
```

```
            stat.incTotalRollbackFail();
        }
    }
    ……
    ……
    }
}
```

非常清楚，如果是 commit 状态，它其实称为“确定提交状态”，则继续提交，调用 businessActivityManager.finish()方法；如果是 init 状态，称为“确定回滚状态”，则继续回滚，调用 businessActivityManager.abort()方法。此时便完成了 recover 工作。注意注释中指明的，此时对于恢复端，只存在 local 和 mix 两种模式，在对 ActionCreatedPatternState 传参时需要注意。

07-30 补充：

这里考虑一种情况：

> 同库+Local 模式下，二阶段提交成功后。本地 Activity 的状态为 C。然后将 Action 记录和 Activity 记录放到异步待删除的线程中执行。如果在删除前，Recover 捞到了这个状态为 C 的记录，会不会重复提交？

经查看，代码中在删除记录时并没有锁，锁了也没法删了，那到底是怎么来控制这种情况的呢？经过一番查看源码和讨论，得出如下结论：

在 BusinessActivityRecoverServiceImpl.java 中，recover 捞取 Activity 时调用的 findPendingBusinessActivities()方法：

```
// 从指定的DB里捞数据
TddlHelper.setCurrent(dbIdentify);
List<BusinessActivityRecord> activities = new ArrayList<BusinessActivityRecord>();
if (recoverRollbackRecordOnly) {
    activities =
businessActivityStore.findPendingBusinessActivitiesForRollbackOnly(reconverLimit);
} else {
    activities = businessActivityStore.findPendingBusinessActivities(reconverLimit);
}
```

在其对应的具体 sql 语句中，有一个重要的限制：

```
<!-- mapped statement for IbatisBusinessActivityDAO.findForRecovery -->
<select id="MS-BUSINESS-ACTIVITY-FIND-FOR-RECOVERY" resultMap="RM-BUSINESS-ACTIVITY">
<![CDATA[
    select tx_id,state,account_trans_state,gmt_create,gmt_modified,context from
beyond_business_activity where ((state NOT IN ('S', 'A')) AND (gmt_create < (sysdate - (1 / 1440)))
AND (rownum < #value#))
]]>
</select>
```

(gmt_create < (sysdate - (1 / 1440)指明了，捞取的数据必须是 1 分钟之前的（我的 xts-core 版本是 4.4.10，指定 1 分钟；在 4.3.2 中是指定 30 秒，这里也解答了和@潇桐 之前讨论时的疑问），也就是说，如果二阶段顺利执行，成功 COMMIT 了，那么异步线程应该在 30 秒/1 分钟之内把 Action 以及 Activity 记录给删除了，如果还没删除，说明有问题，可能挂掉了，这时候需要 recover；如果没挂掉，真的删除了 30 秒/1 分钟，那么就可能出现重复提交了。这时的话需要幂等控制来防止重复提交。

4.5 小结

这一章中对 XTS 源码的分析主要是走了一遍几个主要流程的主要代码，并没有深入去研究实现机理，只是梳理了一下几个重要流程中的细节，可以帮助理解。然而 XTS 背后编写了大量的代码来完成两阶段提交这个过程，并且通过 recover 机制确保一致性，很多东西都可以深入去研究，并且 XTS 本身还在不断演化中，版本不断升级，也有新功能加入到其中，比如关于账户和国际支付的部分本文并没有去研究，因此同学们可以继续深挖 XTS 源码背后的实现机理，彻底把 XTS 搞明白。