

#分布式事物解决方案

第一章 事务简介

- 1.1事物特性(acid)
- 1.2事物隔离级别

第二章 分布式事务简介

- 2.1分布式CAP理论
 - 2.1.1一致性
 - 2.1.2可用性
 - 2.1.3分区容错
- 2.2Base理论
- 2.3柔性事务和刚性事务

第三章 分布式事务解决方案

- 3.1常见解决方案
- 3.2什么是XA接口
- 3.3什么是JTA
- 3.4 2PC两段提交
- 3.5 3PC三段提交
 - 3.5.1 CanCommit阶段
 - 3.5.2 PreCommit阶段
 - 3.5.3 doCommit阶段
 - 3.5.4 2PC与3PC的区别
- 3.6 TCC

第四章 MQ分布式事务

- 4.1 本地事务
- 4.2 分布式事务—两阶段提交协议
- 4.3 使用消息队列来避免分布式事务
 - 4.3.1 如何可靠保存凭证（消息）
 - 4.3.1.1 业务与消息耦合的方式
 - 4.3.1.2 业务与消息解耦方式
 - 4.3.2 如何解决消息重复投递的问题

第五章 LCN分布式事务

- 5.1LCN 事务控制原理
- 5.2LCN事务模式
- 5.3部署tx-manager协调者
 - 5.3.1 部署Eureka服务
 - 5.3.2 官网下载tx-manager
 - 5.3.3 启动事务协调者
- 5.4库存服务还原库存接口
 - 5.4.1 初始数据准备
 - 5.4.2 创建库存服务项目
 - 5.4.3库存服务持久层开发
 - 5.4.4库存服务服务层开发
 - 5.4.5库存服务表现层开发
 - 5.4.6测试库存服务
- 5.5订单服务取消订单接口
 - 5.5.1创建订单服务项目
 - 5.5.2订单服务持久层开发
 - 5.5.3 开发基于Feign的消费者
 - 5.5.4开发订单服务服务层
 - 5.5.5开发订单服务表现层

第一章 事务简介

1.1事物特性(acid)

原子性 (A)

所谓的原子性就是说，在整个事务中的所有操作，要么全部完成，要么全部不做，没有中间状态。对于事务在执行中发生错误，所有的操作都会被回滚，整个事务就像从没被执行过一样。

一致性 (C)

事务的执行必须保证系统的一致性，就拿转账为例，A有500元，B有300元，如果在一个事务里A成功转给B50元，那么不管并发多少，不管发生什么，只要事务执行成功了，那么最后A账户一定是450元，B账户一定是350元。

隔离性 (I)

所谓的隔离性就是说，事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

持久性 (D)

所谓的持久性，就是说一单事务完成了，那么事务对数据所做的变更就完全保存在了数据库中，即使发生停电，系统宕机也是如此。

这种特性 简称 刚性事物

1.2事物隔离级别

更新丢失：两事务同时更新，一个失败回滚覆盖另一个事务的更新。

脏读：事务T1读取到事务T2修改了但是还未提交的数据，之后事务T2又回滚其更新操作，导致事务T1读到的是脏数据。

不可重复读：事务T1读取某个数据后，事务T2对其做了修改，当事务T1再次读该数据时得到与前一次不同的值。

虚读（幻读）：事务T1读取在读取某范围数据时，事务T2又插入一条数据，当事务T1再次数据这个范围数据时发现不一样了，出现了一些“幻影行”。

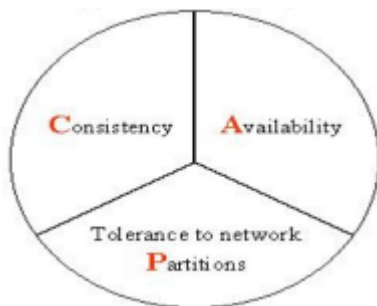
不可重复读和脏读的区别：脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

幻读和不可重复读的异同：都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

第二章 分布式事务简介

分布式事务就是指事务的参与者、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。以上是百度百科的解释，简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。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。

2.1 分布式CAP理论



CAP由[Eric Brewer](#)在2000年PODC会议上提出[1][2]，是Eric Brewer在Inktomi[3]

期间研发搜索引擎、分布式web缓存时得出的关于数据一致性(consistency)、服务可用性(availability)、分区容错性(partition-tolerance)的猜想：

- 数据一致性(consistency)：如果系统对一个写操作返回成功，那么之后的读请求都必须读到这个新数据；如果返回失败，那么所有读操作都不能读到这个数据，对调用者而言数据具有强一致性(strong consistency) (又叫原子性 atomic、线性一致性 linearizable consistency)[5]
- 服务可用性(availability)：所有读写请求在一定时间内得到响应，可终止、不会一直等待
- 分区容错性(partition-tolerance)：在网络分区的情况下，被分隔的节点仍能正常对外服务

2.1.1 一致性

一致性指“all nodes see the same data at the same time”，即更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致。分布式的一致性 对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。如果能容忍后续的部分或者全部访问不到，则是弱一致性。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

2.1.2 可用性

可用性指“Reads and writes always succeed”，即服务一直可用，而且是正常响应时间。对于一个可用性的分布式系统，每一个非故障的节点必须对每一个请求作出响应。也就是，该系统使用的任何算法必须最终终止。当同时要求分区容忍性时，这是一个很强的定义：即使是严重的网络错误，每个请求必须终止。好的可用性主要是指系统能够很好的为用户服务，不出现用户操作失败或者访问超时等用户体验不好的情况。可用性通常情况下可用性和分布式数据冗余，负载均衡等有着很大的关联。

2.1.3 分区容错

分区容错性指“the system continues to operate despite arbitrary message loss or failure of part of the system”，即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。分区容错性和扩展性紧密相关。在分布式应用中，可能因为一些分布式的原因导致系统无法正常运转。好的分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。比如 现在的分布式系统中

有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，或者是机器之间有网络异常，将分布式系统分隔成几个部分，各个部分还能维持分布式系统的运作，这样就具有好的分区容错性。

2.2 Base理论

BASE理论是指，Basically Available（基本可用）、Soft-state（软状态/柔性事务）、Eventual Consistency（最终一致性）。是基于CAP定理演化而来，是对CAP中一致性和可用性权衡的结果。核心思想：即使无法做到强一致性，但每个业务根据自身的特点，采用适当的方式来使系统达到最终一致性。

1、基本可用：指分布式系统在出现故障的时候，允许损失部分可用性，保证核心可用。但不等价于不可用。比如：搜索引擎0.5秒返回查询结果，但由于故障，2秒响应查询结果；网页访问过大时，部分用户提供降级服务，等。

2、软状态：软状态是指允许系统存在中间状态，并且该中间状态不会影响系统整体可用性。即允许系统在不同节点间副本同步的时候存在延时。

3、最终一致性：

系统中的所有数据副本经过一定时间后，最终能够达到一致的状态，不需要实时保证系统数据的强一致性。最终一致性是弱一致性的一种特殊情况。BASE理论面向的是大型高可用可扩展的分布式系统，通过牺牲强一致性来获得可用性。ACID是传统数据库常用的概念设计，追求强一致性模型。

ACID，指数据库事务正确执行的四个基本要素的缩写。包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

Base理论为柔性事务

2.3 柔性事务和刚性事务

柔性事务满足BASE理论（基本可用，最终一致）

刚性事务满足ACID理论

本文主要围绕分布式事务当中的柔性事务的处理方式进行讨论。

柔性事务分为

1. 两阶段型
2. 补偿型
3. 异步确保型
4. 最大努力通知型几种。由于支付宝整个架构是SOA架构，因此传统单机环境下数据库的ACID事务满足不了分布式环境下的业务需要，以上几种事务类似就是针对分布式环境下业务需要设定的。

第三章 分布式事务解决方案

3.1 常见解决方案

1. 分布式事务解决方案 可以使用全局事务2pc（两段提交协议）、3pc（三段提交协议），tcc补偿机制、提供回滚接口、分布式数据库
2. LCN 核心采用3PC+TCC补偿机制

3.2 什么是XA接口

XA-eXtended Architecture 在事务中意为分布式事务 XA由协调者(coordinator, 一般为transaction manager)和参与者(participants,一般在各个资源上有各自的resource manager)共同完成。在MySQL中, XA事务有两种。

3.3什么是JTA

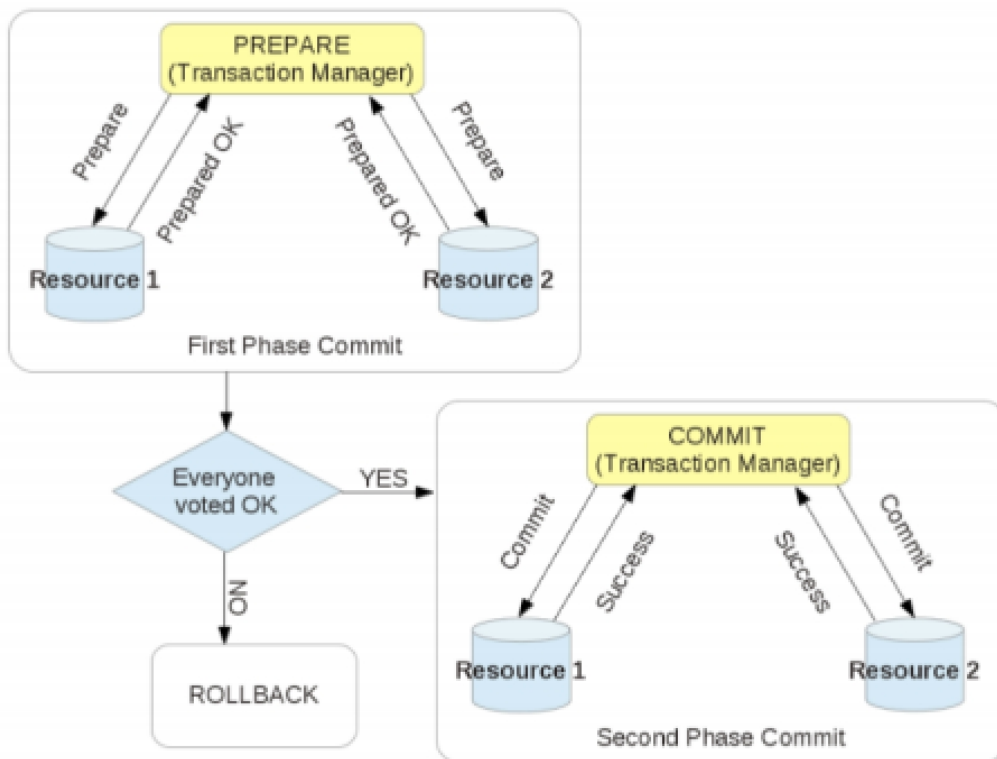
作为java平台上事务规范JTA (Java Transaction API) 也定义了对XA事务的支持, 实际上, JTA是基于XA架构上建模的, 在JTA 中, 事务管理器抽象为javax.transaction.TransactionManager接口, 并通过底层事务服务(即JTS)实现。像很多其他的java规范一样, JTA仅仅定义了接口, 具体的实现则是由供应商(如J2EE厂商)负责提供, 目前JTA的实现主要由以下几种:

1.J2EE容器所提供的JTA实现(JBoss) 2.独立的JTA实现:如JOTM, Atomikos.这些实现可以应用在那些不使用J2EE应用服务器的环境里用以提供分布事务保证。如Tomcat,Jetty以及普通的java应用。

3.4 2PC两段提交

所谓的两个阶段是指: 第一阶段: 准备阶段(投票阶段)和第二阶段: 提交阶段(执行阶段)。

XA一般由两阶段完成, 称为two-phase commit(2PC)。阶段一为准备阶段, 即所有的参与者准备执行事务并锁住需要的资源。参与者ready时, 向transaction manager汇报自己已经准备好。(通知每一方开启事务,然后执行相关操作,操作完成后返回 ok 给协调者) 阶段二为提交阶段。当transaction manager确认所有参与者都ready后, 向所有参与者发送commit命令。如下图所示:



XA的性能问题 XA的性能很低。一个数据库的事务和多个数据库间的XA事务性能对比可发现, 性能差10倍左右。因此要尽量避免XA事务, 例如可以将数据写入本地, 用高性能的消息系统分发数据。或使用数据库复制等技术。只有在这些都无法实现, 且性能不是瓶颈时才应该使用XA。

3.5 3PC三段提交

三阶段提交（Three-phase commit），也叫三阶段提交协议（Three-phase commit protocol），是二阶段提交（2PC）的改进版本。



与两阶段提交不同的是，三阶段提交有两个改动点。

1、引入超时机制。同时在协调者和参与者中都引入超时机制。 2、在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC把2PC的准备阶段再次一分为二，这样三阶段提交就有CanCommit、PreCommit、DoCommit三个阶段。

3.5.1 CanCommit阶段

3PC的CanCommit阶段其实和2PC的准备阶段很像。协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

1.事务询问 协调者向参与者发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

2.响应反馈 参与者接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No

3.5.2 PreCommit阶段

协调者根据参与者的反应情况来决定是否可以记性事务的PreCommit操作。根据响应情况，有以下两种可能。

假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务的预执行。

1.发送预提交请求 协调者向参与者发送PreCommit请求，并进入Prepared阶段。

2.事务预提交 参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

3.响应反馈 如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

1.发送中断请求 协调者向所有参与者发送abort请求。

2.中断事务 参与者收到来自协调者的abort请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

3.5.3 doCommit阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

执行提交

- 1.发送提交请求 协调者接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。
- 2.事务提交 参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
- 3.响应反馈 事务提交完之后，向协调者发送Ack响应。
- 4.完成事务 协调者接收到所有参与者的ack响应之后，完成事务。

中断事务 协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

- 1.发送中断请求 协调者向所有参与者发送abort请求
- 2.事务回滚 参与者接收到abort请求之后，利用其在阶段二记录的undo信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
- 3.反馈结果 参与者完成事务回滚之后，向协调者发送ACK消息
- 4.中断事务 协调者接收到参与者反馈的ACK消息之后，执行事务的中断。

在doCommit阶段，如果参与者无法及时接收到来自协调者的doCommit或者reboot请求时，会在等待超时之后，会继续进行事务的提交。（其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了PreCommit请求，那么协调者产生PreCommit请求的前提条件是他在第二阶段开始之前，收到所有参与者的CanCommit响应都是Yes。（一旦参与者收到了PreCommit，意味他知道大家其实都同意修改了）所以，一句话概括就是，当进入第三阶段时，由于网络超时等原因，虽然参与者没有收到commit或者abort响应，但是他有理由相信：成功提交的几率很大。）

3.5.4 2PC与3PC的区别

相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

3.6 TCC

TCC (Try-Confirm-Cancel)，则是将业务逻辑分成try、confirm/cancel两个阶段执行，具体介绍见[TCC事务机制简介](#)。其事务处理方式为：1、在全局事务决定提交时，调用与try业务逻辑相对应的confirm业务逻辑；2、在全局事务决定回滚时，调用与try业务逻辑相对应的cancel业务逻辑。可见，TCC在事务处理方式上，是很简单的：要么调用confirm业务逻辑，要么调用cancel逻辑

第四章 MQ分布式事务

采用时效性高的MQ，由对方订阅消息并监听，有消息时自动触发事件。采用定时轮询扫描的方式，去检查消息表的数据。

如从支付宝转账1万块钱到余额宝，这是日常生活的一件普通小事，但作为互联网研发人员的职业病，我就思考支付宝扣除1万之后，如果系统挂掉怎么办，这时余额宝账户并没有增加1万，数据就会出现不一致状况了。

上述场景在各个类型的系统中都能找到相似影子，比如在电商系统中，当有用户下单后，除了在订单表插入一条记录外，对应商品表的这个商品数量必须减1吧，怎么保证？！在搜索广告系统中，当用户点击某广告后，除了在点击事件表中增加一条记录外，还得去商家账户表中找到这个商家并扣除广告费吧，怎么保证？！等等，相信大家或多或少都能碰到相似情景。

本质上问题可以抽象为：当一个表数据更新后，怎么保证另一个表的数据也必须要更新成功。

4.1 本地事务

还是以支付宝转账余额宝为例，假设有

- 支付宝账户表：A (id, userId, amount)
- 余额宝账户表：B (id, userId, amount)
- 用户的userId=1;

从支付宝转账1万块钱到余额宝的动作分为两步：

- 1) 支付宝表扣除1万：update A set amount=amount-10000 where userId=1;
- 2) 余额宝表增加1万：update B set amount=amount+10000 where userId=1;

如何确保支付宝余额宝收支平衡呢？有人说这个很简单嘛，可以用事务解决。

```
Begin transaction
update A set amount=amount-10000 where userId=1;
update B set amount=amount+10000 where userId=1;
End transaction
commit;
```

非常正确，如果你使用spring的话一个注解就能搞定上述事务功能。

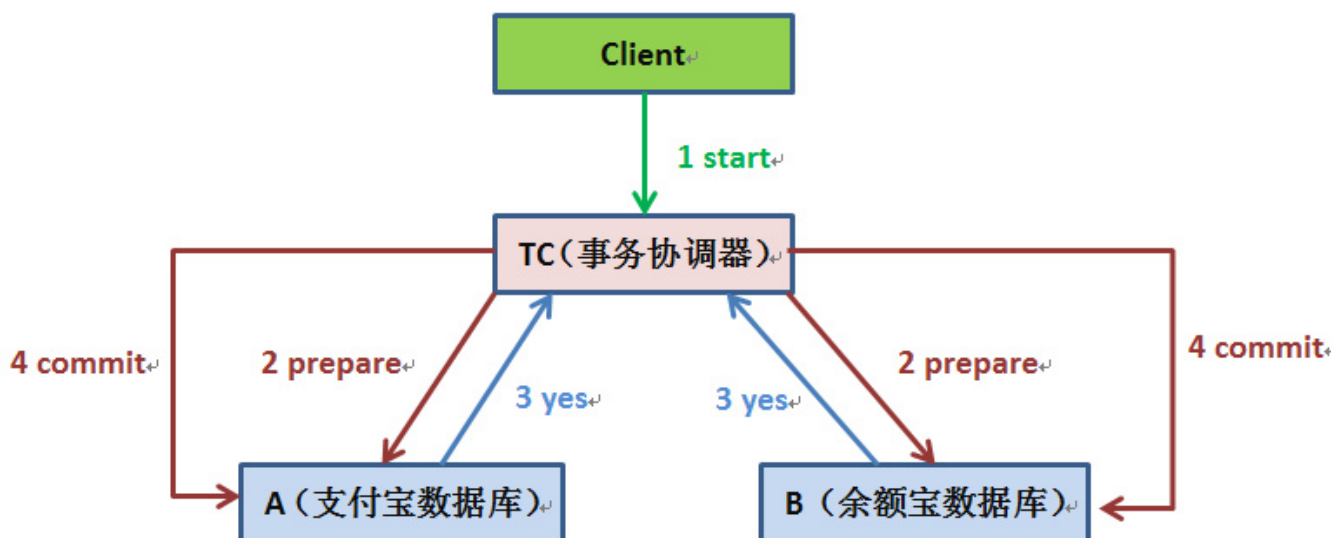
```
@Transactional(rollbackFor=Exception.class)
public void update() {
    updateATable();//更新A表
    updateBTable();//更新B表
}
```

如果系统规模较小，数据表都在一个数据库实例上，上述本地事务方式可以很好地运行，但是如果系统规模较大，比如支付宝账户表和余额宝账户表显然不会在同一个数据库实例上，他们往往分布在不同的物理节点上，这时本地事务已经失去用武之地。

既然本地事务失效，分布式事务自然就登上舞台。

4.2 分布式事务—两阶段提交协议

两阶段提交协议（Two-phase Commit, 2PC）经常被用来实现分布式事务。一般分为协调器C和若干事务执行者Si两种角色，这里的事务执行者就是具体的数据库，协调器可以和事务执行器在一台机器上。



- 1) 我们的应用程序 (client) 发起一个开始请求到TC;
- 2) TC先将消息写到本地日志, 之后向所有的Si发起消息。以支付宝转账到余额宝为例, TC给A的prepare消息是通知支付宝数据库相应账目扣款1万, TC给B的prepare消息是通知余额宝数据库相应账目增加1w。为什么在执行任务前需要先写本地日志, 主要是为了故障后恢复用, 本地日志起到现实生活中凭证 的效果, 如果没有本地日志 (凭证), 出问题容易死无对证;
- 3) Si收到消息后, 执行具体本机事务, 但不会进行commit, 如果成功返回, 不成功返回。同理, 返回前都应把要返回的消息写到日志里, 当作凭证。
- 4) TC收集所有执行器返回的消息, 如果所有执行器都返回yes, 那么给所有执行器发生送commit消息, 执行器收到commit后执行本地事务的commit操作; 如果有任一个执行器返回no, 那么给所有执行器发送abort消息, 执行器收到abort消息后执行事务abort操作。

注: TC或Si把发送或接收到的消息先写到日志里, 主要是为了故障后恢复用。如某一Si从故障中恢复后, 先检查本机的日志, 如果已收到, 则提交, 如果则回滚。如果是, 则再向TC询问一下, 确定下一步。如果什么都没有, 则很可能在阶段Si就崩溃了, 因此需要回滚。

现如今实现基于两阶段提交的分布式事务也没那么困难了, 如果使用java, 那么可以使用开源软件atomikos(<http://www.atomikos.com/>)来快速实现。

不过但凡使用过的上述两阶段提交的同学都可以发现性能实在是太差, 根本不适合高并发的系统。为什么?

- 1) 两阶段提交涉及多次节点间的网络通信, 通信时间太长!
- 2) 事务时间相对于变长了, 锁定的资源的时间也变长了, 造成资源等待时间也增加好多! 正是由于分布式事务存在很严重的性能问题, 大部分高并发服务都在避免使用, 往往通过其他途径来解决数据一致性问题。

4.3 使用消息队列来避免分布式事务

如果仔细观察生活的话, 生活的很多场景已经给了我们提示。

比如在北京很有名的姚记炒肝点了炒肝并付了钱后, 他们并不会直接把你点的炒肝给你, 而是给你一张小票, 然后让你拿着小票到出货区排队去取。为什么他们要将付钱和取货两个动作分开呢? 原因很多, 其中一个很重要的原因是为了使他们接待能力增强 (并发量更高)。

还是回到我们的问题, 只要这张小票在, 你最终是能拿到炒肝的。同理转账服务也是如此, 当支付宝账户扣除1万后, 我们只要生成一个凭证 (消息) 即可, 这个凭证 (消息) 上写着“让余额宝账户增加 1万”, 只要这个凭证 (消息) 能可靠保存, 我们最终是可以拿着这个凭证 (消息) 让余额宝账户增加1万的, 即我们能依靠这个凭证 (消息) 完成最终一致性。

4.3.1 如何可靠保存凭证（消息）

有两种方法：

4.3.1.1 业务与消息耦合的方式

支付宝在完成扣款的同时，同时记录消息数据，这个消息数据与业务数据保存在同一数据库实例里（消息记录表表名为message）。

```
Begin transaction
update A set amount=amount-10000 where userId=1;
insert into message(userId, amount,status) values(1, 10000, 1);
End transaction
commit;
```

上述事务能保证只要支付宝账户里被扣了钱，消息一定能保存下来。

当上述事务提交成功后，我们通过实时消息服务将此消息通知余额宝，余额宝处理成功后发送回复成功消息，支付宝收到回复后删除该条消息数据。

4.3.1.2 业务与消息解耦方式

本方式适合RocketMQ等带事务的消息中间件。

上述保存消息的方式使得消息数据和业务数据紧耦合在一起，从架构上看不够优雅，而且容易诱发其他问题。为了解耦，可以采用以下方式。

- 1) 支付宝在扣款事务提交之前，向实时消息服务请求发送消息，实时消息服务只记录消息数据，而不真正发送，只有消息发送成功后才会提交事务；
- 2) 当支付宝扣款事务被提交成功后，向实时消息服务确认发送。只有在得到确认发送指令后，实时消息服务才真正发送该消息；
- 3) 当支付宝扣款事务提交失败回滚后，向实时消息服务取消发送。在得到取消发送指令后，该消息将不会被发送；
- 4) 对于那些未确认的消息或者取消的消息，需要有一个消息状态确认系统定时去支付宝系统查询这个消息的状态并进行更新。为什么需要这一步骤，举个例子：假设在第2步支付宝扣款事务被成功提交后，系统挂了，此时消息状态并未被更新为“确认发送”，从而导致消息不能被发送。

优点：消息数据独立存储，降低业务系统与消息系统间的耦合；

缺点：一次消息发送需要两次请求；业务处理服务需要实现消息状态回查接口。

4.3.2 如何解决消息重复投递的问题

还有一个很严重的问题就是消息重复投递，以我们支付宝转账到余额宝为例，如果相同的消息被重复投递两次，那么我们余额宝账户将会增加2万而不是1万了。

为什么相同的消息会被重复投递？比如余额宝处理完消息msg后，发送了处理成功的消息给支付宝，正常情况下支付宝应该要删除消息msg，但如果支付宝这时候悲剧的挂了，重启后一看消息msg还在，就会继续发送消息msg。

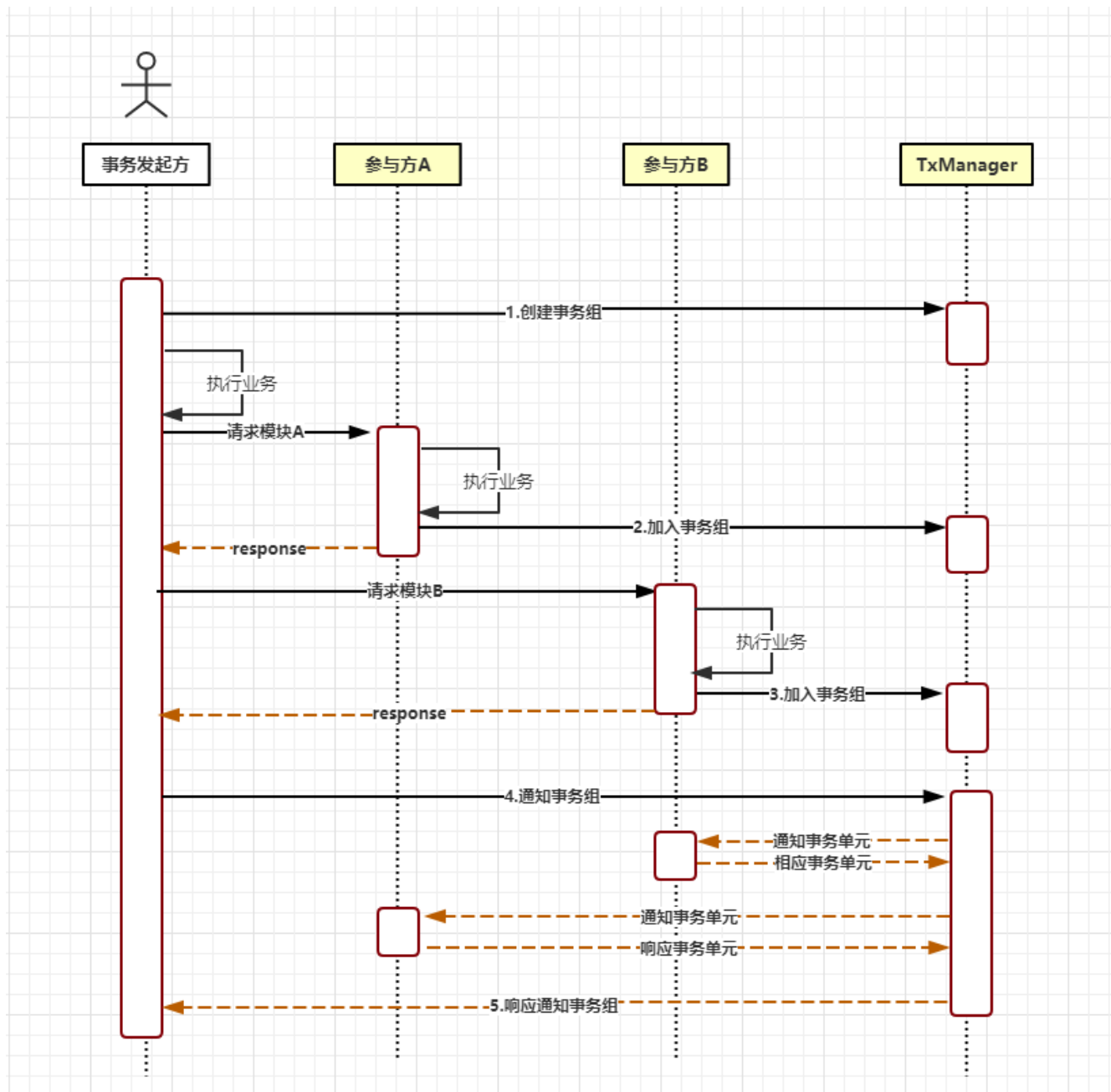
解决方法很简单，在余额宝这边增加消息应用状态表（message_apply），通俗来说就是个账本，用于记录消息的消费情况，每次来一个消息，在真正执行之前，先去消息应用状态表中查询一遍，如果找到说明是重复消息，丢弃即可，如果没找到才执行，同时插入到消息应用状态表（同一事务）。

```
foreach msg in queue
Begin transaction
  select count(*) as cnt from message_apply where msg_id=msg.msg_id;
  if cnt==0 then
    update B set amount=amount+10000 where userId=1;
    insert into message_apply(msg_id) values(msg.msg_id);
  End transaction
commit;
```

第五章 LCN分布式事务

5.1 LCN 事务控制原理

1. TX-LCN由两大模块组成, **TxClient**、**TxManager**, TxClient作为模块的依赖框架, 提供TX-LCN的标准支持, TxManager作为分布式事务的控制方。事务发起方或者参与反都由TxClient端来控制。 (**简单来说就是单独部署一套TxManager模块来实现事务管理, TxClient就是我们自己的服务系统**)
2. 原理图如下:



5.2LCN事务模式

原理介绍

1. LCN模式是通过**代理Connection**的方式实现对**本地事务**的操作，然后在由TxManager统一协调控制事务。当本地事务提交回滚或者关闭连接时将会执行假操作，该代理的连接将由LCN连接池管理。

模式特点

1. 该模式对代码的**嵌入性为低**。
2. 该模式仅限于**本地存在连接对象**且可通过连接对象控制事务的模块。
3. 该模式下的事务提交与回滚是由本地事务方控制，对于**数据一致性**上有较高的保障。
4. 该模式缺陷在于**代理的连接需要随事务发起方一共释放连接**，增加了连接占用的时间。

5.3部署tx-manager协调者

5.3.1 部署Eureka服务

因为tx-manager协调者需要向注册中心注册服务。

这里的注册中心不配置用户名和密码

```
server.port=8761
security.basic.enabled=false
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
#服务的地址,注册中心的地址
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

5.3.2 官网下载tx-manager

<https://github.com/codingapi/tx-lcn/releases>

v4.1.0
5f8f55a
Verified


v4.1.0

 1991wangliang released this on 11 Mar 2018 · 566 commits to master since this release

1. 简化操作，注意。发起方需要添加isStart=true条件。
2. 修改issues发现的问题。
3. 修改Connection的获取连接方式。
4. 对应非数据库操作时无效使用no-db的扩展。

关于4.1.0的操作可参考wiki与demo下4.1.0的分支代码。

▼ Assets 2

 [Source code \(zip\)](#)

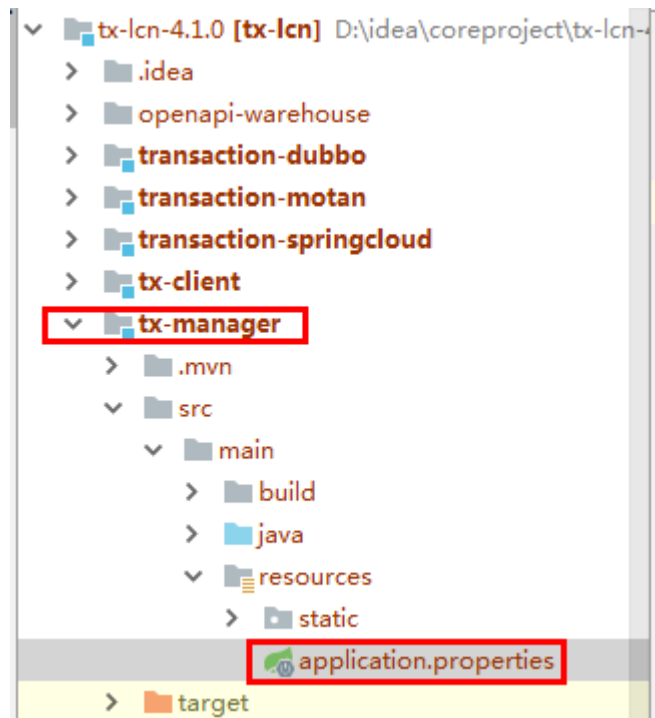
 [Source code \(tar.gz\)](#)

<https://github.com/codingapi/tx-lcn/archive/v4.1.0.zip>

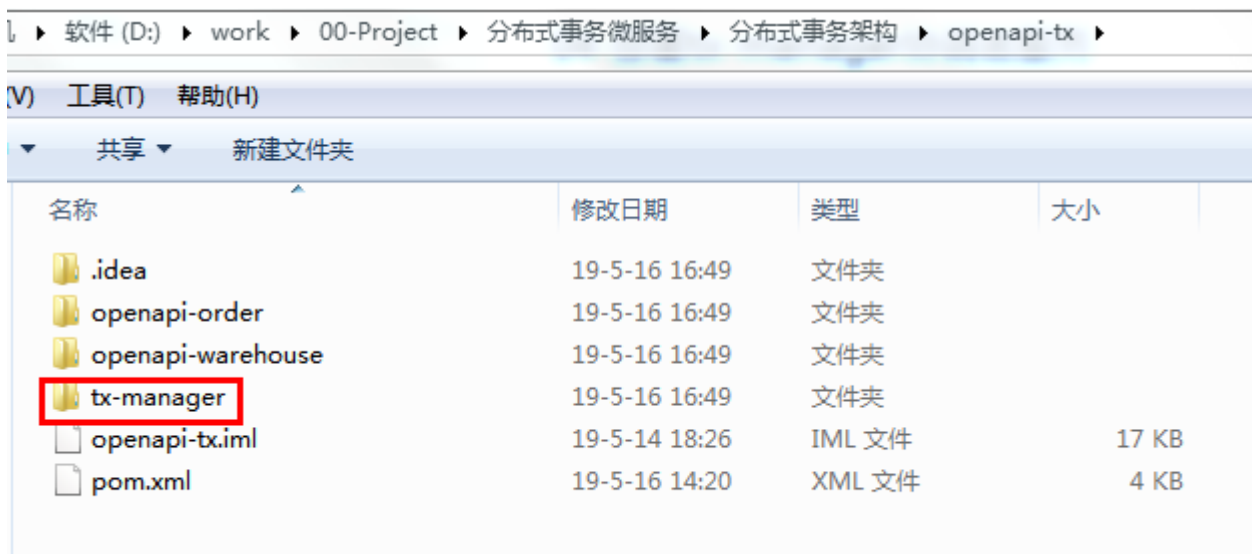
5.3.3 启动事务协调者

首先需要启动Eureka

打开源码项目：



这里直接使用项目的架构:



修改配置:application.properties

eureka地址和redis地址需要修改为自己的:

修改配置:


```
eureka.client.service-url.defaultZone=http://qianfeng1.qfjava.cn:9201/eureka
eureka.instance.prefer-ip-address=true
```

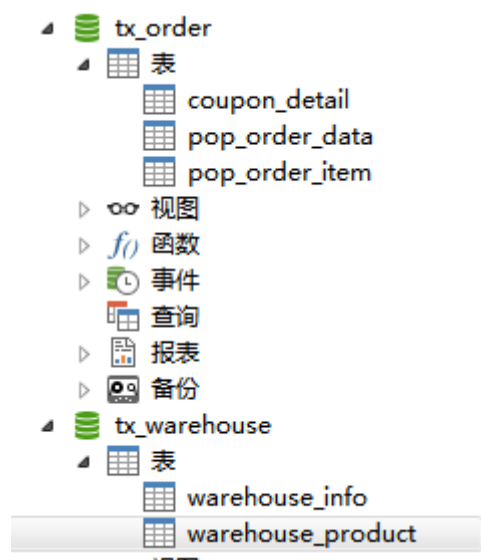
```
#redis
#redis主机地址
spring.redis.host=114.242.146.109
#redis主机端口
spring.redis.port=8400
#redis链接密码 密码根据服务器的情况是否有
#spring.redis.password=redis001
```

5.4 库存服务还原库存接口

5.4.1 初始数据准备

订单服务数据库tx_order
库存服务数据库tx_warehouse

需要创建2个数据库，分别加入初始数据。



5.4.2 创建库存服务项目

openapi-warehouse,可在原有项目结构的基础上新增:

pom.xml环境继承了父类openapi-tx

```
<dependencies>
  <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
  </dependency>
  <dependency>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>druid</artifactId>
<version>1.0.19</version>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.46</version>
</dependency>
<!--工具类 start-->
<dependency>
<groupId>commons-lang</groupId>
<artifactId>commons-lang</artifactId>
<version>2.6</version>
</dependency>
<!--工具类 end-->
<!-- ini 工具体类 start -->
<dependency>
<groupId>org.ini4j</groupId>
<artifactId>ini4j</artifactId>
<version>0.5.4</version>
</dependency>
<!-- 日志包 -->
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-api</artifactId>
<version>1.7.7</version>
</dependency>
<!-- slf4j日志框架和log4j 转换包 -->
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-log4j12</artifactId>
<version>1.7.7</version>
</dependency>
</dependencies>
```

需要修改配置文件:application.properties

1. 数据库地址
2. 注册中心的地址
3. 事务协调者的地址(入口)

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://114.242.146.109:8300/tx_warehouse?characterEncoding=UTF-8&useSSL=false&serverTimezone=UTC
```

```
spring.datasource.username=root
spring.datasource.password=qishimeiyoumima
spring.datasource.initialize=true
#扫描映射文件
mybatis.mapper-locations=classpath:mapper/*Mapper.xml

spring.application.name=openapi-warehouse
server.port = 8082
eureka.client.service-url.defaultZone=http://114.242.146.109:9201/eureka/
#txmanager地址
tm.manager.url=http://127.0.0.1:8899/tx/manager/
logging.level.com.qianfeng=debug
```

修改入口程序：

Eureka服务的提供者

加载Druid的数据源.

```
package com.qianfeng.openapi;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.feign.EnableFeignClients;
import org.springframework.context.annotation.Bean;
import org.springframework.core.env.Environment;

import javax.sql.DataSource;

@EnableAutoConfiguration
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class WarehouseApplication {
    public static void main(String[] args) {
        SpringApplication.run(WarehouseApplication.class, args);
    }
    @Autowired
    private Environment env;
    @Bean
    public DataSource dataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUrl(env.getProperty("spring.datasource.url"));
        dataSource.setUsername(env.getProperty("spring.datasource.username")); //用户名
        dataSource.setPassword(env.getProperty("spring.datasource.password")); //密码
        dataSource.setInitialSize(2);
        dataSource.setMaxActive(20);
        dataSource.setMinIdle(0);
    }
}
```

```
dataSource.setMaxWait(60000);
dataSource.setValidationQuery("SELECT 1");
dataSource.setTestOnBorrow(false);
dataSource.setTestWhileIdle(true);
dataSource.setPoolPreparedStatements(false);
return dataSource;
}
}
```

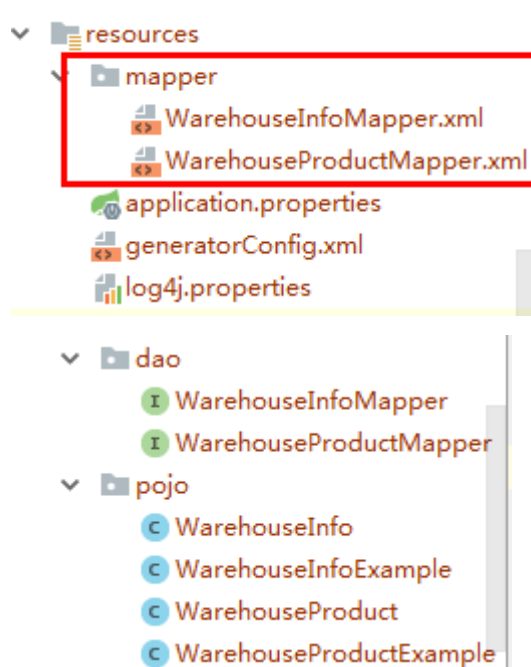
5.4.3 库存服务持久层开发

插件生成持久层

generatorConfig.xml:修改为自己仓库中的jar包地址与数据库地址账户

```
<build>
  <plugins>
    <plugin>
      <groupId>org.mybatis.generator</groupId>
      <artifactId>mybatis-generator-maven-plugin</artifactId>
      <version>1.3.5</version>
      <configuration>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
      </configuration>
    </plugin>
  </plugins>
</build>
```

生成结果:



注意: DAO中加@Mapper注解

5.4.4 库存服务服务层开发

根据skuld修改库存表 (warehouse_product) 中的商品数量current_cnt

接口:

```
package com.qianfeng.openapi.service;

public interface HouserProductService {
    public int updateHouse(Long skuId,int cnt);
}
```

实现类:

```
package com.qianfeng.openapi.service.impl;

import com.codingapi.tx.annotation.ITxTransaction;
import com.qianfeng.openapi.dao.WarehouseProductMapper;
import com.qianfeng.openapi.pojo.WarehouseProduct;
import com.qianfeng.openapi.pojo.WarehouseProductExample;
import com.qianfeng.openapi.service.HouserProductService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
public class HouserProductServiceImpl implements HouserProductService,ITxTransaction {

    @Autowired
    private WarehouseProductMapper warehouseProductMapper;
    //必须加上本地事务，不然后面回滚无法回滚
    @Transactional
    public int updateHouse(Long skuId,int cnt) {
        WarehouseProductExample example = new WarehouseProductExample();
        WarehouseProductExample.Criteria criteria = example.createCriteria();
        criteria.andProductIdEqualTo(skuId.intValue());
        List<WarehouseProduct> warehouseProducts =
warehouseProductMapper.selectByExample(example);
        WarehouseProduct warehouseProduct = warehouseProducts.get(0);
        //在原有值上加1
        warehouseProduct.setCurrentCnt(warehouseProduct.getCurrentCnt()+cnt);
        int i = warehouseProductMapper.updateByPrimaryKeySelective(warehouseProduct);
        return i;
    }
}
```

说明:

- 1, 根据skuld先查询出来记录, 再修改具体字段
- 2, 需要实现事务协调者的接口ITxTransaction

开发事务协调者获取具体地址的实现类:

```
package com.qianfeng.openapi.service.impl;

import com.codingapi.tx.config.service.TxManagerTxUrlService;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
@Service
public class TxManagerTxUrlServiceImpl implements TxManagerTxUrlService{
    @Value("${tm.manager.url}")
    private String url;//读取配置文件
    @Override
    public String getTxUrl() {
        System.out.println("load tm.manager.url ");
        return url;//返回事务协调者的真实地址
    }
}
```

说明：从配置文件application.properties获取具体的值

5.4.5 库存服务表现层开发

```
package com.qianfeng.openapi.control;

import com.qianfeng.openapi.service.HouserProductService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

@RestController
public class HouseProductControl {

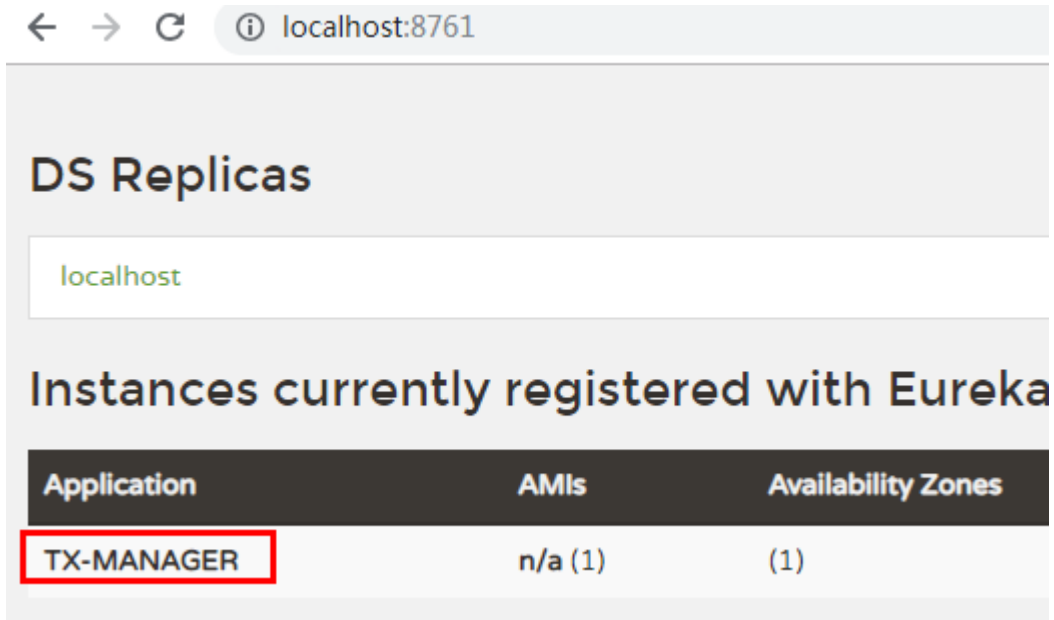
    @Autowired
    private HouserProductService houserProductService;

    @RequestMapping("/house/productres/{skuId}/{cnt}")
    public int updateProduct(@PathVariable("skuId") Long skuId , @PathVariable("cnt")
Integer cnt){
        Map<String,Object> map = new HashMap<>();
        int i = houserProductService.updateHouse(skuId,cnt);
        return i;
    }
}
```

说明：这里是微服务的提供者，直接返回修改的结果i，消费者[订单服务]会调用这个接口，直接拿到结果。

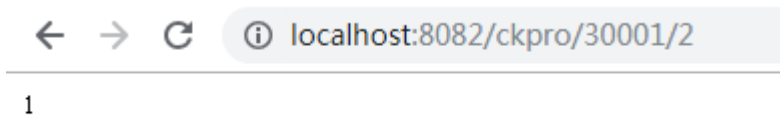
5.4.6 测试库存服务

1. 开启注册中心
2. 开启事务协调者
3. 开启库存服务

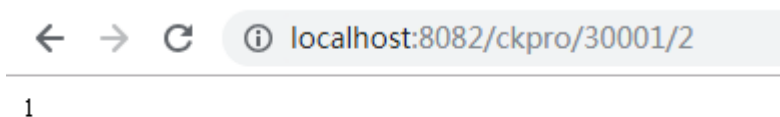


测试库存接口

<http://localhost:8082/ckpro/30001/2>



数据库变化:



5.5 订单服务取消订单接口

5.5.1 创建订单服务项目

openapi-order

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
  </dependency>
</dependencies>
```

```
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.19</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
</dependency>
<!--工具类 start-->
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.6</version>
</dependency>
<!--工具类 end-->
<!-- ini 工具体类 start -->
<dependency>
    <groupId>org.ini4j</groupId>
    <artifactId>ini4j</artifactId>
    <version>0.5.4</version>
</dependency>
<!-- 日志包 -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.7</version>
</dependency>
<!-- slf4j日志框架和log4j 转换包 -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.7</version>
</dependency>
<!--swagger2 依赖-->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
```

```
<version>2.7.0</version>
</dependency>
</dependencies>
```

说明：多加入swagger在线文档的依赖

修改配置文件application.properties

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://114.242.146.109:8300/tx_order?characterEncoding=UTF-
8&useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=qishimeiyoumima
spring.datasource.initialize=true
#扫描映射文件
mybatis.mapper-locations=classpath:mapper/*Mapper.xml
## 关于springcloud-hystrix机制 http://www.jianshu.com/p/b8d21248c9b1
#hystrix.command.default.execution.isolation.strategy= SEMAPHORE
#hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=5000
spring.application.name =openapi-order
server.port = 8081
eureka.client.service-url.defaultZone=http://114.242.146.109:9201/eureka/
#Ribbon的负载均衡策略
ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule
ribbon.MaxAutoRetriesNextServer=0
#txmanager地址
tm.manager.url=http://127.0.0.1:8899/tx/manager/
logging.level.com.qianfeng=debug
#在线api配置
swagger.basePackage=com.qianfeng.openapi.controller
swagger.title=分布式事务相关接口文档
swagger.description=提供订单相关的服务
swagger.contact=qianfeng
swagger.version=v1.0
```

修改入口程序：

```
package com.qianfeng.openapi;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.feign.EnableFeignClients;
import org.springframework.context.annotation.Bean;
import org.springframework.core.env.Environment;

import javax.sql.DataSource;

@EnableAutoConfiguration
```

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class OrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
    @Autowired
    private Environment env;
    @Bean
    public DataSource dataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUrl(env.getProperty("spring.datasource.url"));
        dataSource.setUsername(env.getProperty("spring.datasource.username")); //用户名
        dataSource.setPassword(env.getProperty("spring.datasource.password")); //密码
        dataSource.setInitialSize(10);
        dataSource.setMaxActive(50);
        dataSource.setMinIdle(0);
        dataSource.setMaxWait(60000);
        dataSource.setValidationQuery("SELECT 1");
        dataSource.setTestOnBorrow(false);
        dataSource.setTestWhileIdle(true);
        dataSource.setPoolPreparedStatements(false);
        return dataSource;
    }
}
```

swagger配置类:

```
package com.qianfeng.openapi.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

/**
 * 在线文档配置
 *
 * 访问网址: 1.http://ip:端口/api 或 2.http://ip:端口/swagger-ui.html
 *
 * @author: qianfeng
 * @version: v1.0
 * @date: 2019/3/9 14:31
 */
```

```
@Configuration
@EnableSwagger2
public class SwaggerConfig extends WebMvcConfigurerAdapter {

    @Value("${swagger.basePackage}")
    private String basePackage;// 扫描controller的包名

    @Value("${swagger.title}")
    private String title;// 在线文档的标题

    @Value("${swagger.description}")
    private String description;// 在线文档的描述

    @Value("${swagger.contact}")
    private String contact;// 联系人

    @Value("${swagger.version}")
    private String version;// 文档版本

    @Bean
    public Docket createRedisApi() {
        return new Docket(DocumentationType.SWAGGER_2).apiInfo(apiInfo()).select();// 函数返回
        一个ApiSelectorBuilder实例用来控制哪些接口暴露给Swagger来展现

        .apis(RequestHandlerSelectors.basePackage(basePackage)).paths(PathSelectors.any()).build();
    }

    /**
     * apiInfo()用来创建该Api的基本信息
     * 这些基本信息会展现在文档页面中
     */
    private ApiInfo apiInfo() {
        return new
        ApiInfoBuilder().title(title).description(description).contact(contact).version(version)
            .build();
    }

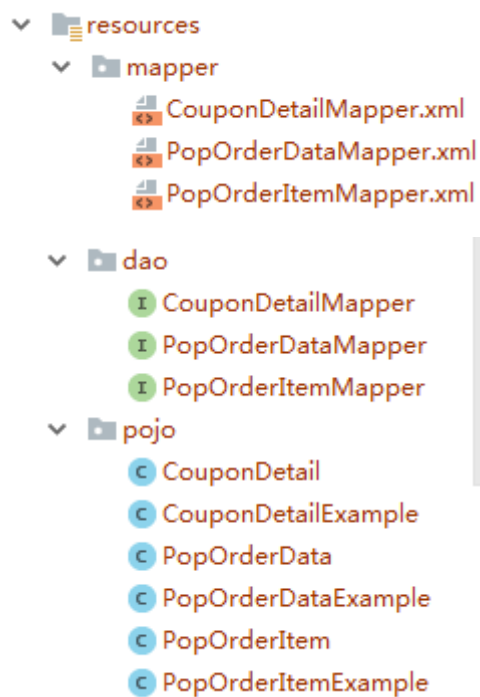
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addRedirectViewController("/v1/api-docs", "/v1/api-docs");
        // 添加服务api访问文档url
        registry.addRedirectViewController("/api", "/swagger-ui.html");
    }
}
```

5.5.2 订单服务持久层开发

插件生成持久层 generatorConfig.xml:修改为自己仓库中的jar包地址与数据库地址账户

```
<build>
  <plugins>
    <plugin>
      <groupId>org.mybatis.generator</groupId>
      <artifactId>mybatis-generator-maven-plugin</artifactId>
      <version>1.3.5</version>
      <configuration>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
      </configuration>
    </plugin>
  </plugins>
</build>
```

生成结果：



注意：DAO中加@Mapper注解

5.5.3 开发基于Feign的消费者

定义Feign接口


```
package com.qianfeng.openapi.client;

import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
//对应服务提供者在Eureka中的名称
@FeignClient(value = "openapi-warehouse", fallback = HouseFeignClientHystric.class)
public interface HouseFeignClient {
    //和服务提供者在HouseProductControl定义的接口——对应
    @RequestMapping("/house/productres/{skuId}/{cnt}")
    public int updateProduct(@PathVariable("skuId") Long skuId , @PathVariable("cnt")
    Integer cnt);
}
```

回调实现:

```
package com.qianfeng.openapi.client;

import org.springframework.stereotype.Component;

@Component
public class HouseFeignClientHystric implements HouseFeignClient {
    @Override
    public int updateProduct(Long skuId, Integer cnt) {
        System.out.println("进入断路器.....");
        throw new RuntimeException("修改库存失败.");
    }
}
```

5.5.4开发订单服务服务层

定义接口:

```
package com.qianfeng.openapi.service;

import com.qianfeng.openapi.pojo.PopOrderData;

import java.util.Map;

public interface PopOrderDataService {
    //扩展一个认为异常回滚属性exp
    public int updateStatus(PopOrderData pojo,String exp);
}
```

分布式事务实现:

```
package com.qianfeng.openapi.service.impl;

import com.codingapi.tx.annotation.TxTransaction;
import com.qianfeng.openapi.client.HouseFeignClient;
```

```
import com.qianfeng.openapi.dao.PopOrderDataMapper;
import com.qianfeng.openapi.dao.PopOrderItemMapper;
import com.qianfeng.openapi.pojo.PopOrderData;
import com.qianfeng.openapi.pojo.PopOrderItem;
import com.qianfeng.openapi.pojo.PopOrderItemExample;
import com.qianfeng.openapi.service.PopOrderDataService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Service
public class PopOrderDataServiceImpl implements PopOrderDataService {
    @Autowired
    private PopOrderDataMapper popOrderDataMapper;
    @Autowired
    private PopOrderItemMapper popOrderItemMapper;
    @Autowired
    private HouseFeignClient houseFeignClient;

    private Logger logger = LoggerFactory.getLogger(PopOrderDataServiceImpl.class);

    @Override
    @TxTransaction(isStart = true)//分布式事务
    @Transactional//本地事务
    public int updateStatus(PopOrderData pojo,String exp) {
        int i = popOrderDataMapper.updateByPrimaryKeySelective(pojo);
        Long id = pojo.getId();
        PopOrderData popOrderData = popOrderDataMapper.selectByPrimaryKey(id);
        Long orderId = popOrderData.getOrderId();
        PopOrderItemExample example = new PopOrderItemExample();
        PopOrderItemExample.Criteria criteria = example.createCriteria();
        criteria.andOrderIdEqualTo(orderId);
        List<PopOrderItem> popOrderItems = popOrderItemMapper.selectByExample(example);
        PopOrderItem popOrderItem = popOrderItems.get(0);
        Long skuId = popOrderItem.getSkuId();
        int updateProduct = houseFeignClient.updateProduct(skuId, 1);
        if("1".equals(exp)) {
            //人为制造异常
            int v = 100 / 0;
        }
        return i+updateProduct;
    }
}
```

说明:

- 1, 修改**订单服务**订单状态
- 2, 根据订单主键ID逐步得到skuld

3, 基于SpringCloud的Feign调用**库存服务**openapi-warehouse获取返回结果

4, 引入分布式事务注解@Transactional(isStart = true)处理订单服务的库和库存服务的库

开发事务协调者获取具体地址的实现类:

```
package com.qianfeng.openapi.service.impl;

import com.codingapi.tx.config.service.TxManagerTxUrlService;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
@Service
public class TxManagerTxUrlServiceImpl implements TxManagerTxUrlService{
    @Value("${tm.manager.url}")
    private String url;
    @Override
    public String getTxUrl() {
        System.out.println("load tm.manager.url ");
        return url;
    }
}
```

获取请求事务协调者实现类:

```
package com.qianfeng.openapi.service.impl;

import com.codingapi.tx.netty.service.TxManagerHttpRequestService;
import com.lorne.core.framework.utils.http.HttpUtils;
import org.springframework.stereotype.Service;
@Service
public class TxManagerHttpRequestServiceImpl implements TxManagerHttpRequestService{
    @Override
    public String httpGet(String url) {
        System.out.println("httpGet-start");
        String res = HttpUtils.get(url);
        System.out.println("httpGet-end");
        return res;
    }
    @Override
    public String httpPost(String url, String params) {
        System.out.println("httpPost-start");
        String res = HttpUtils.post(url,params);
        System.out.println("httpPost-end");
        return res;
    }
}
```

5.5.5开发订单服务表现层

```
package com.qianfeng.openapi.controller;

import com.qianfeng.openapi.pojo.PopOrderData;
```

```
import com.qianfeng.openapi.service.PopOrderDataService;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

@RestController
@Api(description = "订单管理api", value = "订单管理")
public class OrderController {
    private final static Logger log = LoggerFactory.getLogger(OrderController.class);
    @Autowired
    private PopOrderDataService popOrderDataService;
    @ApiOperation(value = "退货接口")
    @RequestMapping("/cancel/order/{id}/{exp}")
    public Map<String, Object> cancelOrder(@ApiParam(name = "id", value = "订单主键", required = true) @PathVariable("id") Long id,
                                           @ApiParam(name = "exp", value = "人为异常", required = false) @PathVariable("exp") String exp){
        log.info("参数id:{}", id);
        Map<String, Object> map = new HashMap<>();
        PopOrderData pojo = new PopOrderData();
        pojo.setId(id);
        pojo.setState(6L); //状态为6表示取消订单
        int i = popOrderDataService.updateStatus(pojo, exp);
        if(i >= 2){
            map.put("code", 0);
            map.put("msg", "退货成功");
        } else {
            map.put("code", -1);
            map.put("msg", "退货失败");
        }
        return map;
    }
}
```

说明:

- 1, 加入swagger在线文档的注解
- 2, 扩展字段exp可选, 值为1表示会生成人为异常。

5.5.6 分布式事务测试

- 1, 开启Eureka服务
- 2, 开启事务协调者服务tx-manager

3,开启库存服务openapi-warehouse（含服务发布者）

4,开启订单服务（含服务消费者）

swagger测试:

<http://localhost:8081/api>

分布式事务相关接口文档

提供订单相关的服务

Created by qianfeng

order-controller : 订单管理api

DELETE	/cancel/order/{id}/{exp}
GET	/cancel/order/{id}/{exp}
HEAD	/cancel/order/{id}/{exp}
OPTIONS	/cancel/order/{id}/{exp}

点击get方式:

输入参数: 第一次人为异常值不输入; 第二次测试输入1

Parameters

Parameter	Value	Description
id	<input type="text" value="1"/>	订单主键
exp	<input type="text" value="1"/>	人为异常

Response Messages

HTTP Status Code	Reason	Response Model
401	Unauthorized	
403	Forbidden	
404	Not Found	

[Try it out!](#) [Hide Response](#)

数据库中值确认:

退货成功:

订单状态修改成功

对象		* 无标题 @tx_order (qfdb) - ...		warehouse_product @tx_wa...	
		开始事务		备注 筛选 排序 导入 导出	
wp_id	product_id	w_id	current_cnt	ock_cnt	in_transit_cnt
1	30001	1	1000	0	

库存还原成功：

对象		* 无标题 @tx_order (qfdb) - ...		warehouse_product @tx_wa...	
		开始事务		备注 筛选 排序 导入 导出	
wp_id	product_id	w_id	current_cnt	ock_cnt	in_transit_cnt
1	30001	1	1000	0	

5.6学生练习

完成分布式事务案例:取消订单还原库存接口