

第1章 分布式事务

课程安排

第一篇章：

分布式事务基础知识

分布式事务的解决方案分析

2PC方案：atomikos

最终一致性方案-事务消息：RocketMQ

第二篇章：

最终一致性方案-本地消息表（seata框架AT模式）

最终一致性方案-TCC补偿：（seata框架TCC模式）

分布式事务解决方案的优劣分析

分布式事务解决方案再分析-秒杀超卖的解决思路

1 关于分布式事务

分布式事务是由本地事务演变而来的，而本地事务，相信很多同学都已经非常熟悉了，下面1.1小节是针对本地事务的一个简述。

1.1 本地事务

1.1.1 事务的概念

事务指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部不成功。从而确保了数据的准确与安全。

1.1.2 事务的四大特性

1) 原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

2) 一致性（Consistency）

事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

例如转账前A有1000，B有1000。转账后A+B也得是2000。

3) 隔离性（Isolation）

事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，每个事务不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

4) 持久性 (Durability)

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

1.1.3 事务的隔离级别

1) 不考虑隔离级别，会出现以下情况：（以下情况全是错误的）

脏读：一个线程中的事务读到了另外一个线程中未提交的数据。

不可重复读：一个线程中的事务读到了另外一个线程中已经提交的update的数据（前后内容不一样）

虚读（幻读）：一个线程中的事务读到了另外一个线程中已经提交的insert的数据（前后条数不一样）

2) 数据库共定义了四种隔离级别：

Serializable：可避免脏读、不可重复读、虚读情况的发生。（串行化）最高

Repeatable read：可避免脏读、不可重复读情况的发生。（有可能发生幻读）第二

Read committed：可避免脏读情况发生。第三

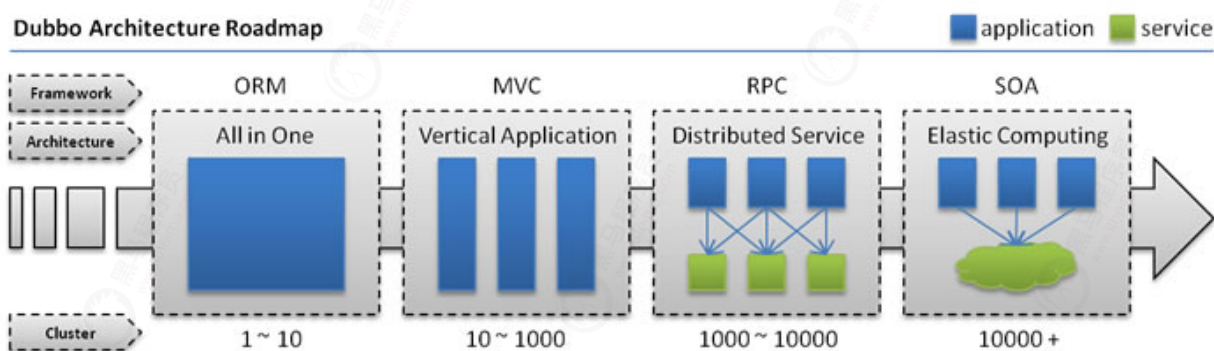
Read uncommitted：最低级别，以上情况均无法保证。（读未提交）最低

注意：级别依次升高，效率依次降低

MySQL的默认隔离级别是：REPEATABLE READ。（Oracle的默认是：READ COMMITTED）

1.1.4 随着系统架构的变化本地事务的问题

在[dubbo官方网址](http://dubbo.apache.org/)上提供了一个系统架构的演变图，它里面展示了从单体架构到SOA架构的演变：



当我们的项目架构不再是All in One时，那么我们的数据库也可能面临着单一数据库不够用的情况。当我们部署了多台数据库之后，新的问题就产生了，由于每台数据库都有独立的本地事务，且事务的隔离性确定了事务之间不能相互干扰。所以，在多数据库下，事务该如何控制呢？这就是我们本篇章要解决的问题。

1.2 分布式事务

1.2.1 分布式事务的概念

分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单来说就是组成事务的各个单元处于不同数据库服务器上。

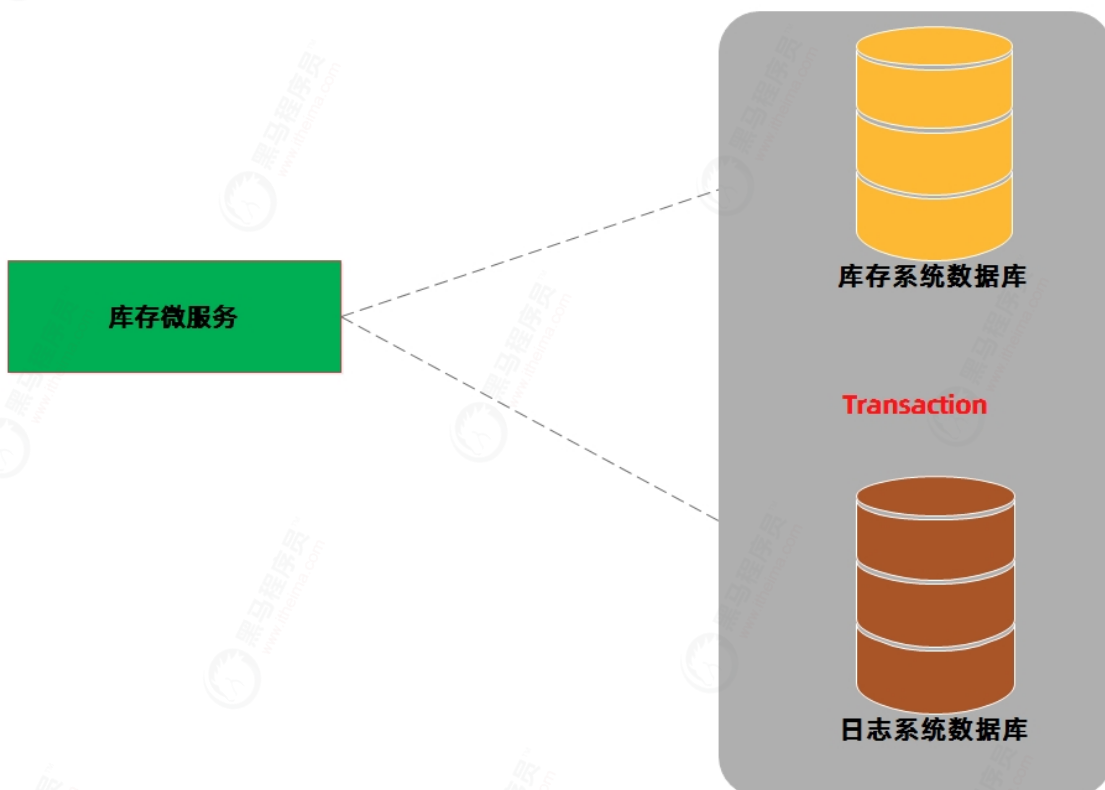
相信同学们都接触过这种场景，手机支付，付款方和收款方的银行账号不是同一家一行，不在同一地域的情况。那么，我们就要保证付款方减去的金额，和收款方增加的金额保持一致。

在我们的实际开发中，分布式事务无处不在，比如，电商系统中的生成订单，账户扣款，减少库存，增加会员积分等等，他们就是组成事务的各个单元，它们要么全部发生，要么全部不发生，从而保证最终一致，数据准确。下一小节我们将分析分布式事务的应用架构（场景）。

1.2.2 分布式事务的应用架构

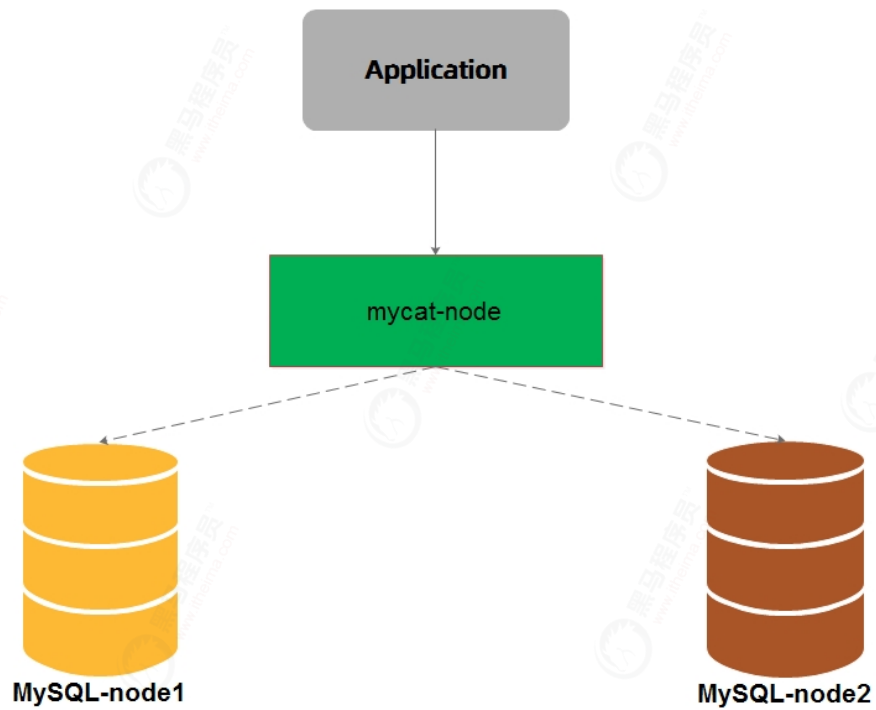
1) 单一服务不同数据库架构

单一服务不同数据库架构



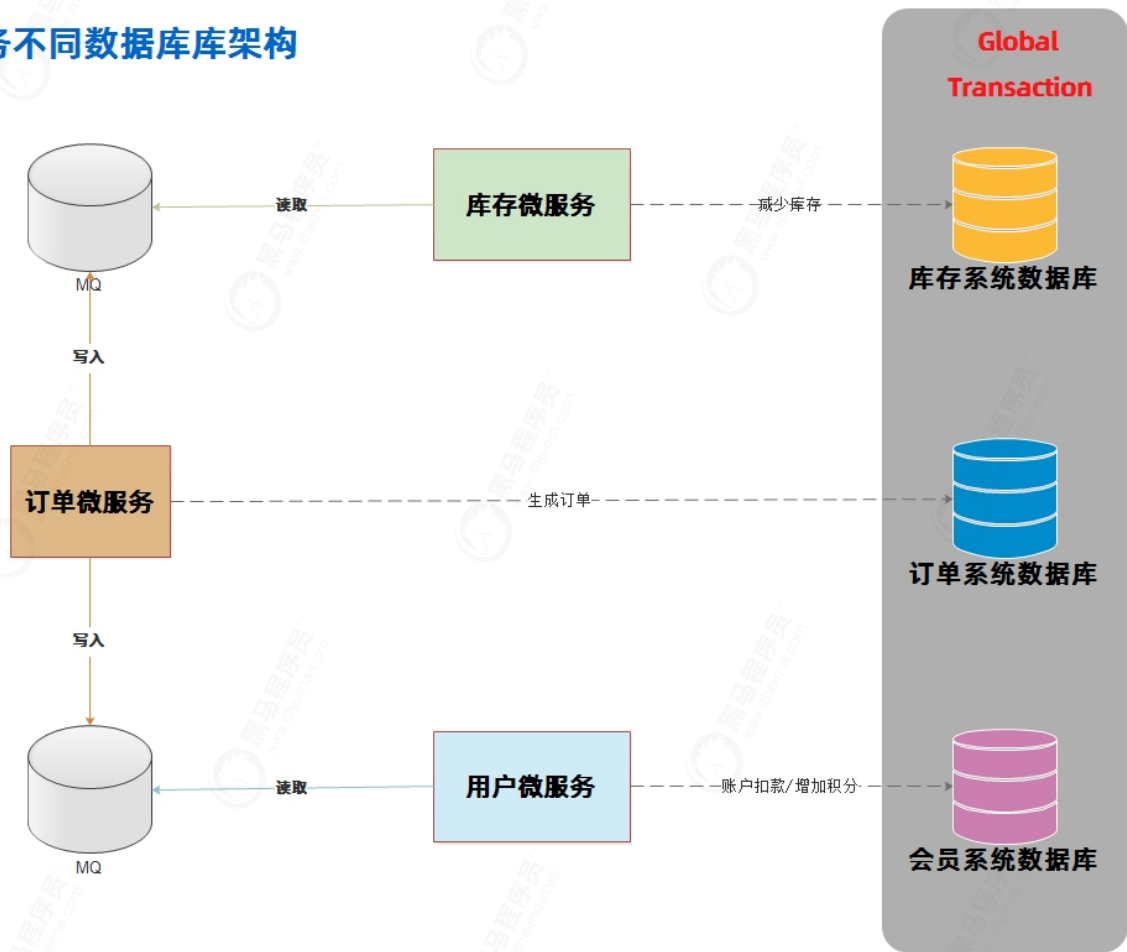
2) 单一服务分库分表架构

单一服务分库分表架构



3) 多服务不同数据库架构

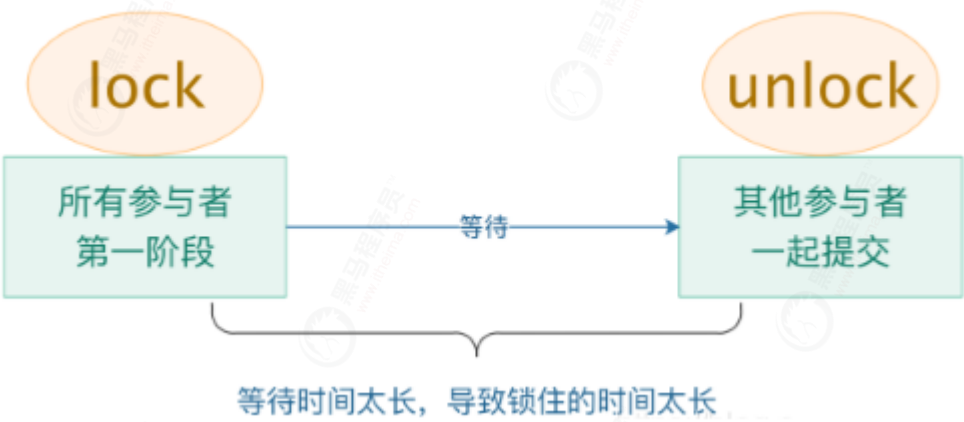
多服务不同数据库架构



1.2.3 分布式事务的解决方案分类

1) 刚性事务

刚性事务指的就是遵循本地事务四大特性 (ACID) 的强一致性事务。它的特点就是强一致性，要求组成事务的各个单元马上提交或者马上回滚，没有时间弹性，要求以同步的方式执行。通常在单体架构项目中应用较多，一般都是企业级应用（或者局域网应用）。例如：生成合同时记录日志，付款成功后生成凭据等等。但是，在时下流行的互联网项目中，这种刚性事务来解决分布式事务就会有许多的弊端。其中最明显的或者说最致命的就是性能问题。如下图所示：



因为某个参与者不能自己提交事务，必须等待所有参与者执行OK了之后，一起提交事务，那么事务锁住的时间就变得非常长，从而导致性能非常低下。基于此我们也就得出了一个结论，阶段越多，性能越差。

2) 柔性事务

柔性事务是针对刚性事务而说的，我们刚才分析了刚性事务，它有两个特点，第一个强一致性，第二个近实时性（NRT）。而柔性事务的特点是不需要立刻马上执行（同步性），且不需要强一致性。它只要满足基本可用和最终一致就可以了。要想真正的明白，需要从BASE理论和CAP理论说起。

1.2.4 CAP理论和BASE理论

1) CAP理论

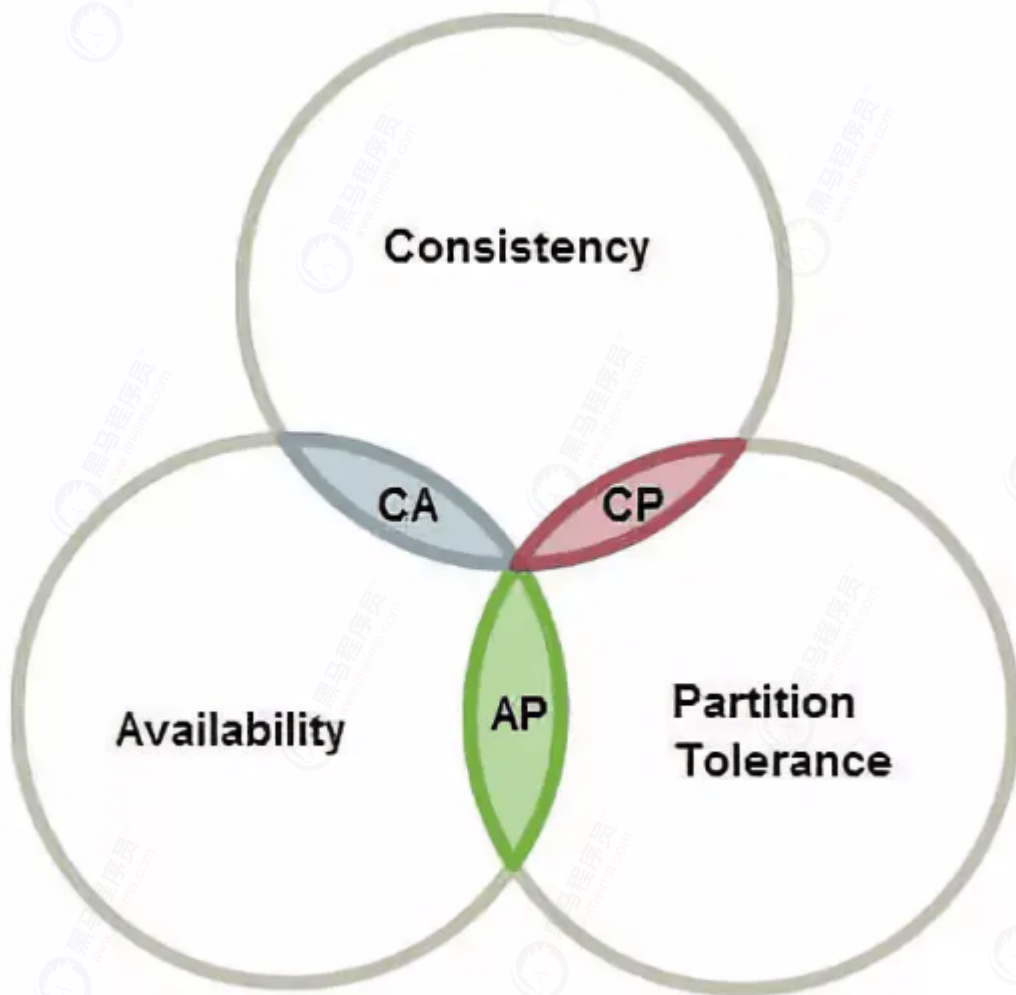
CAP理论，又叫做CAP原则，网上对他的概念描述非常的清晰，且一致。换句话说，我们在网上搜到的CAP理论的描述，基本都是一样的。它的描述是这样的：

CAP指的是在一个分布式系统中，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance）。其中，C,A,P的说明如下：

一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）

可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）

分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。



CAP原则指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。因此在进行分布式架构设计时，必须做出取舍。而对于分布式数据系统，分区容错性是基本要求，否则就失去了价值。因此设计分布式数据系统，就是在一致性和可用性之间取一个平衡。对于大多数web应用，其实并不需要强一致性，因此牺牲一致性而换取高可用性，是目前多数分布式数据库产品的方向。

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）。

1、基本可用 BA：（Basically Available）：

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。简单来说就是基本可用。

2、软状态 S：（Soft State）：

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。简单来说就是状态可以在一段时间内不同步。

3、最终一致性 E：（Eventually Consistent）：

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。简单来说就是在一定的时间窗口内，最终数据达成一致即可。

BASE理论是基于CAP原则演化而来，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。**

3) 再谈刚性事务和柔性事务的比较

事务类型	时间要求	一致性要求	应用类型	应用场景
刚性事务	立即	强一致性	企业级应用（单体架构）	订单/订单项/日志
柔性事务	有时间弹性	最终一致性	互联网应用（分布式架构）	订单/支付/库存

2 分布式事务解决方案

2.1 二阶段提交（2PC）方案详述

2.1.1 DTP标准（DTP模型）

DTP标准，也称为DTP模型。它的全称是：Distributed Transaction Processing，是X/Open提出的一个分布式事务行业标准。X/Open，即现在的open group，是一个独立的组织，[官网地址](#)。X/Open组织主要由各大知名公司或者厂商进行支持，这些组织不光遵循X/Open组织定义的行业技术标准，也参与到标准的制定。下图展示了open group目前主要成员(官网截图)：

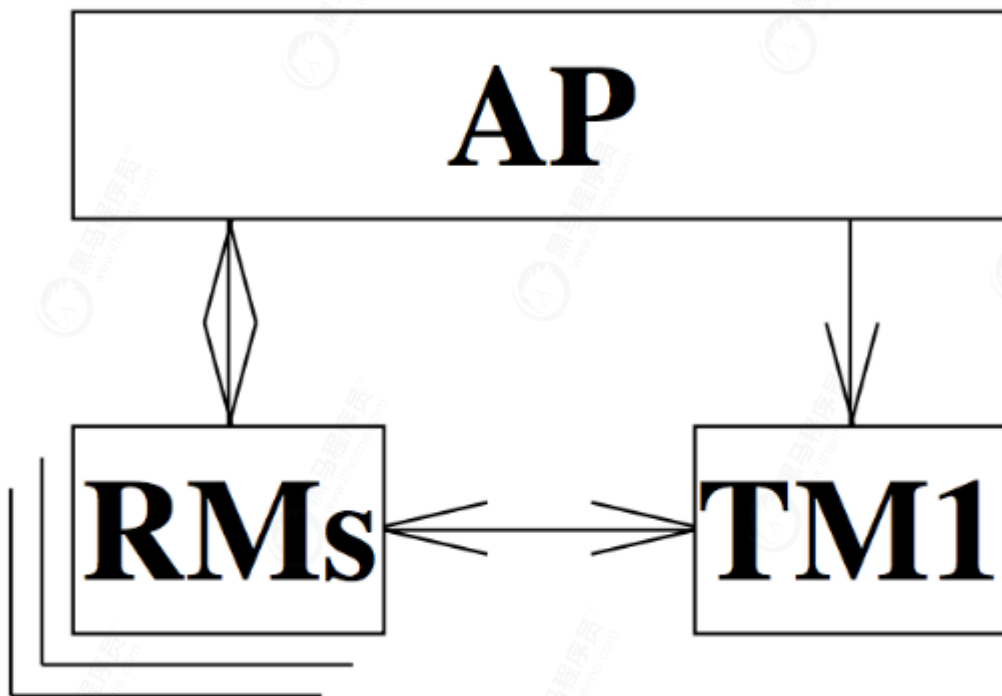


X/Open针对DTP提供了一下参考文档：[Distributed Transaction Processing: Reference Model](#)和[Distributed Transaction Processing: The XA Specification](#)。它里面包含了DTP模型和XA规范。

在DTP规范中，定义了DTP模型和XA规范。DTP模型有5个基本元素组成，他们分别是：

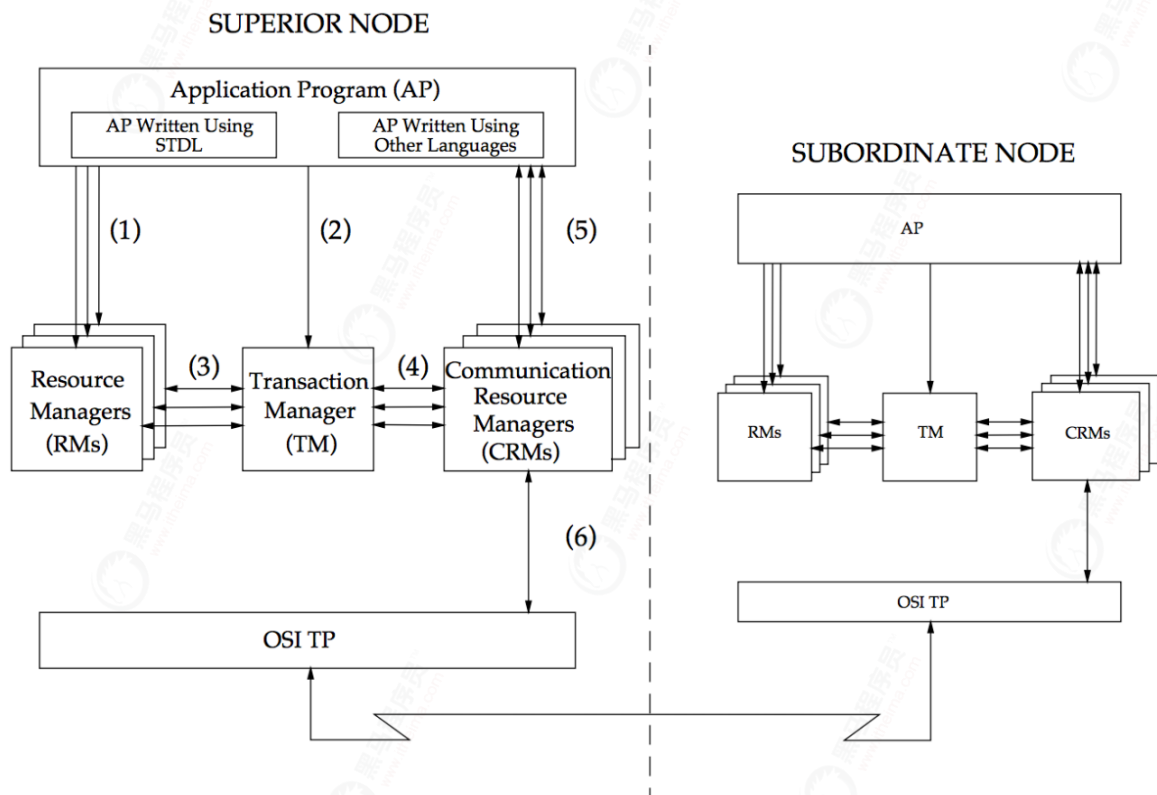
名称	简称	说明
应用程序(Application Program)	AP	用于定义事务边界(即定义事务的开始和结束), 并且在事务边界内对资源进行操作。
资源管理器(Resource Manager)	RM	如数据库、文件系统等, 并提供访问资源的方式。
事务管理器(Transaction Manager)	TM	负责分配事务唯一标识, 监控事务的执行进度, 并负责事务的提交、回滚等。
通信资源管理器(Communication Resource Manager):	CRM	控制一个TM域(TM domain)内或者跨TM域的分布式应用之间的通信。
通信协议(Communication Protocol):	CP	提供CRM提供的分布式应用节点之间的底层通信服务。

一个DTP模型最少需要包含AP, TM和RM三部分组成, 如下图所示:

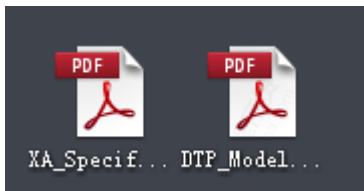


这张图类似于我们之前提到的跨库事务的概念, 即单个应用需要操作多个库。在这里就是一个AP需要操作多个RM上的资源。AP通过TM来声明一个全局事务, 然后操作不同的RM上的资源, 最后通知TM来提交或者回滚全局事务。

特别的, 如果分布式事务需要跨多个应用, 类似于我们前面的提到的分布式事务场景中的服务化, 那么每个模型实例中, 还需要额外的加入一个通信资源管理器CRM。下图中演示了2个模型实例, 如何通过CRM来进行通信:



Tips: 以上内容来源于官方文档。



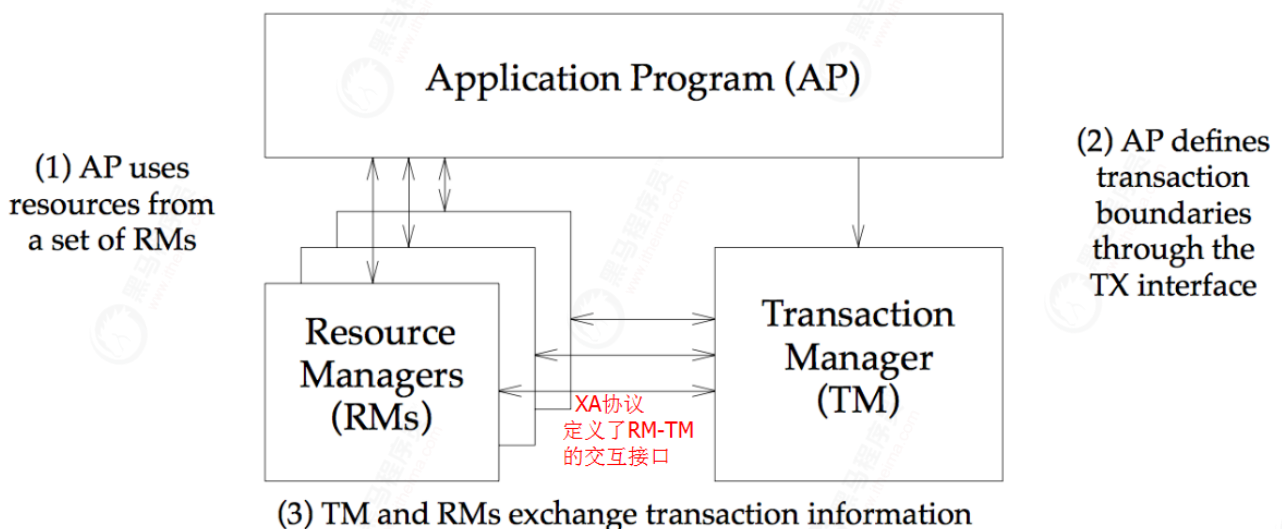
2.1.2 XA协议规范

XA协议是资源管理器（数据库）与事务管理器的接口标准。目前，Oracle、Informix、DB2和Sybase等各大数据库厂家都提供对XA的支持。XA协议采用两阶段提交方式来管理分布式事务。XA接口提供资源管理器与事务管理器之间进行通信的标准接口。XA协议包括两套函数，以xa开头的及以ax开头的。

XA规范中定义的RM 和 TM交互的接口如下图所示：

Name	Description
<i>ax_reg</i>	Register an RM with a TM.
<i>ax_unreg</i>	Unregister an RM with a TM.
<i>xa_close</i>	Terminate the AP's use of an RM.
<i>xa_commit</i>	Tell the RM to commit a transaction branch.
<i>xa_complete</i>	Test an asynchronous <i>xa_</i> operation for completion.
<i>xa_end</i>	Dissociate the thread from a transaction branch.
<i>xa_forget</i>	Permit the RM to discard its knowledge of a heuristically-completed transaction branch.
<i>xa_open</i>	Initialise an RM for use by an AP.
<i>xa_prepare</i>	Ask the RM to prepare to commit a transaction branch.
<i>xa_recover</i>	Get a list of XIDs the RM has prepared or heuristically completed.
<i>xa_rollback</i>	Tell the RM to roll back a transaction branch.
<i>xa_start</i>	Start or resume a transaction branch - associate an XID with future work that the thread requests of the RM.

我们以DTP本地模型实例中，由AP、RMs和TM组成，不需要其他元素。AP、RM和TM之间，彼此都需要进行交互，如下图所示：



但是，XA协议是指定RM和TM的交互规范。在官方文献中有着清晰的介绍，如下图：

The subject of this X/Open specification is interface (3) in the diagram above, the XA interface by which TMs and RMs interact.

For more details on this model and diagram, including detailed definitions of each component, see the referenced **DTP** guide.

XA协议是语言无关，平台无关的。但是，我们都知道开发语言有很多种，针对不同的开发语言，都会定义自己针对XA协议的规范。

2.1.3 JTA

JTA，它是XA协议的Java版本规范。全称是Java Transaction API。它是JavaEE的13个规范之一。[官方网站](#)提供了资源的下载：

Downloads

Download the JTA specification, interfaces, and Javadoc files.

Java Transaction API Specification 1.1 Maintenance Release [Download](#)

Class Files 1.1 [Download](#)

Javadocs 1.1 [Download](#)

Java Transaction API Specification 1.0.1 B Maintenance Release of Maintenance Reviews [Download](#)

Class Files 1.0.1B [Download](#)

Javadocs 1.0.1B [Download](#)

Java Transaction API Specification 1.0.1 [Download](#)

Errata September 27, 2001 in Java Community Process Maintenance Review until November 5, 2001 has closed. Errata has been approved.

Class files and Javadocs which include Errata January 31, 2001 and Errata June 30, 2000:

Class Files 1.0.1a [Download](#)

Javadocs 1.0.1a [Download](#)

Comments may be submitted to jta-feedback@sun.com.

JTA interfaces, specification and Javadoc files: [Download](#)

Resources

Browse Javadocs: [javax.transaction](#) and [javax.transaction.xa](#)

See the Java EE 5 Tutorial section: [JTA Transactions](#)

我们可以下载它的手册，源码和API文档。打开API看到里面的接口，如下图：

[All Classes](#)

Packages

[javax.transaction](#)
[javax.transaction.xa](#)

[javax.transaction](#)

Interfaces

[Status](#)
[Synchronization](#)
[Transaction](#)
[TransactionManager](#)
[TransactionSynchronizationRegistry](#)
[UserTransaction](#)

Exceptions

[HeuristicCommitException](#)
[HeuristicMixedException](#)
[HeuristicRollbackException](#)
[InvalidTransactionException](#)
[NotSupportedException](#)
[RollbackException](#)
[SystemException](#)
[TransactionRequiredException](#)
[TransactionRolledbackException](#)

这里面定义的都是标准，是基于Java语言开发分布式事务二阶段提交的标准。而要想真正的使用，必须使用它的实现。在早期我们都学习过JavaEE容器技术（也有称为服务器或中间件的），像Tomcat，weblogic，websphere，JBoss，Jetty，Resin等等。这里面有轻量级，也有重量级。凡是重量级服务器都是完整实现了JavaEE规范的，也就是说可以直接使用JTA的实现。除了完整的实现了JavaEE规范的容器外，还有单独针对JTA进行实现的，像atomikos，JOTM。

2.1.4 atomikos

atomikos是一款开源的，且实现了JTA规范的分布式事务解决方案，可以在[github](#)，[官方网站](#)上下载它的资料。在github上可以看到它的亮点，如下图：

Highlights

- **Distributed transaction management for Java** - so your Java application gets **instant reliability** without design efforts in your code.
- **Microservices support** - so you can have one global transaction spanning several services.
- **JEE compatible** - so it integrates effortlessly in your **Spring** or **Tomcat** configuration.
- **Light-weight** - so your **microservices** can use it, too.
- **Embeddable in your code** - so you can **test everything in the IDE** and avoid late integration issues at deployment time.
- **OSGi support** - so you can use OSGi containers also.
- **Connection pooling for JDBC and JMS** - so you get maximum performance.
- **Built-in support for Hibernate and JPA** - so you can use your favorite persistence framework.
- **Automatic crash / restart recovery** - so your incomplete distributed transactions are **cleaned up** and your data returns to a consistent state.
- **Cloud-native design** - so your applications are ready for **deployment to your cloud**.
- **Commercial support available** - if you're serious about transactions in your business.

从上面，我们可以看出，它支持java的分布式事务管理，支持微服务，支持JDBC连接池和JMS消息机制。在早期的二阶段提交方式的分布式事务技术选型中，它的出场几率是非常高的。

2.2 TCC补偿型方案

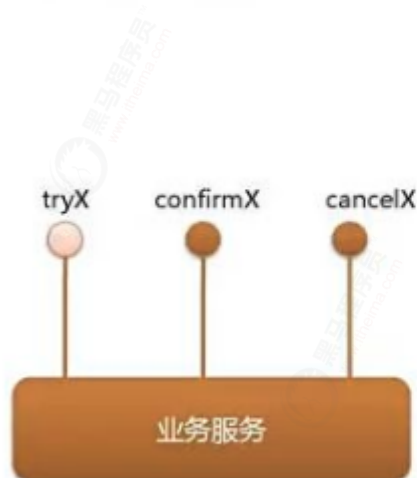
2.2.1 关于TCC

TCC分别指的是Try, Confirm, Cancel。它是补偿型分布式事务解决方案。何为补偿呢？其实我们把TCC这3个部分分别做什么捋清楚，就很容易理解了。首先，我们先来看下它们的主要作用：

Try 阶段主要是对业务系统做检测及资源预留。

Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。

Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。



Try: 尝试执行业务

- 完成所有业务检查(一致性)
- 预留必须业务资源(准隔离性)

Confirm: 确认执行业务

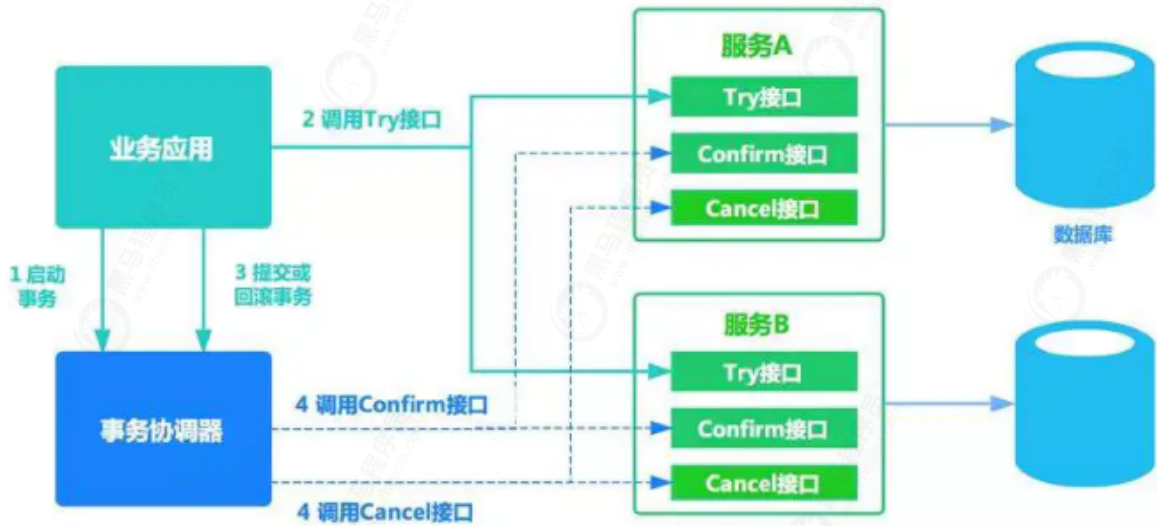
- 真正执行业务
- 不作任何业务检查
- 只使用Try阶段预留的业务资源
- Confirm操作满足幂等性

Cancel: 取消执行业务

- 释放Try阶段预留的业务资源
- Cancel操作满足幂等性

由此，我们可以得出结论，就是在Try阶段进行尝试提交事务，当Try执行OK了，Confirm执行，且默认认为它一定成功。但是当Try提交失败了，则由Cancel处理回滚和资源释放。

2.2.2 TCC流程图



2.2.3 TCC和2PC的优劣分析

两阶段提交



TCC事务的处理流程与2PC两阶段提交做比较，首先TCC是柔性事务，只要符合最终一致性即可。而2PC是刚性事务，它是强一致性的，在任何一个分布式阶段没有返回执行成功或失败的结果时，其事务一直会处于等待状态。并且2PC是利用DTP模型和XA规范，要求数据库支持XA规范，且通常都是在跨库的DB层面。

而TCC则在应用层面的处理，需要通过自己编写逻辑代码来实现补偿。它的优势在于，可以让应用自己定义数据操作的粒度，使得降低锁冲突、提高吞吐量成为可能。而不足之处则在于对应用的侵入性非常强，业务逻辑的每个分支都需要实现try、confirm、cancel三个操作。此外，其实现难度也比较大，需要按照网络状态、系统故障等不同的失败原因实现不同的回滚策略。

2.3 最终一致性方案

2.3.1 本地消息表

1) 简述

这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆成本地事务进行处理，这种思路是来源于ebay。它和MQ事务消息的实现思路都是一样的，都是利用MQ通知不同的服务实现事务的操作。不同的是，针对消息队列的信任情况，分成了两种不同的实现。本地消息表它是对消息队列的稳定性处于不信任的态度，认为消息可能会出现丢失，或者消息队列的运行网络会出现阻塞，于是在数据库中建立一张独立的表，用于存放事务执行的状态，配合消息队列实现事务的控制。

2) 优缺点

优点：一种非常经典的实现，避免了分布式事务，实现了最终一致性。

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

2.3.2 MQ事务消息

1) 简述

有一些第三方的MQ是支持事务消息的，比如RocketMQ，ActiveMQ，他们支持事务消息的方式也是类似于采用的二阶段提交。但是有一些常用的MQ也不支持事务消息，比如 RabbitMQ 和 Kafka 都不支持。

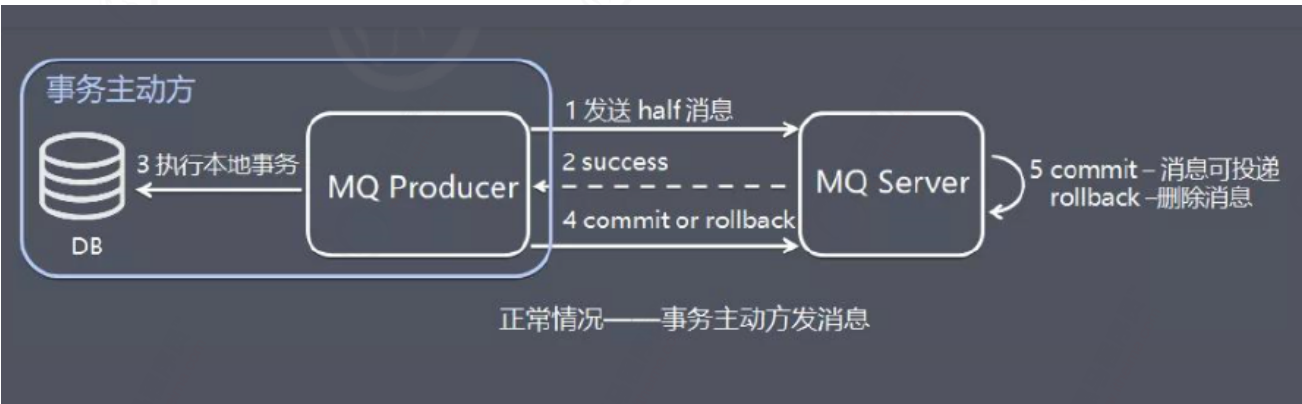
以阿里的 RocketMQ 中间件为例，其思路大致为：

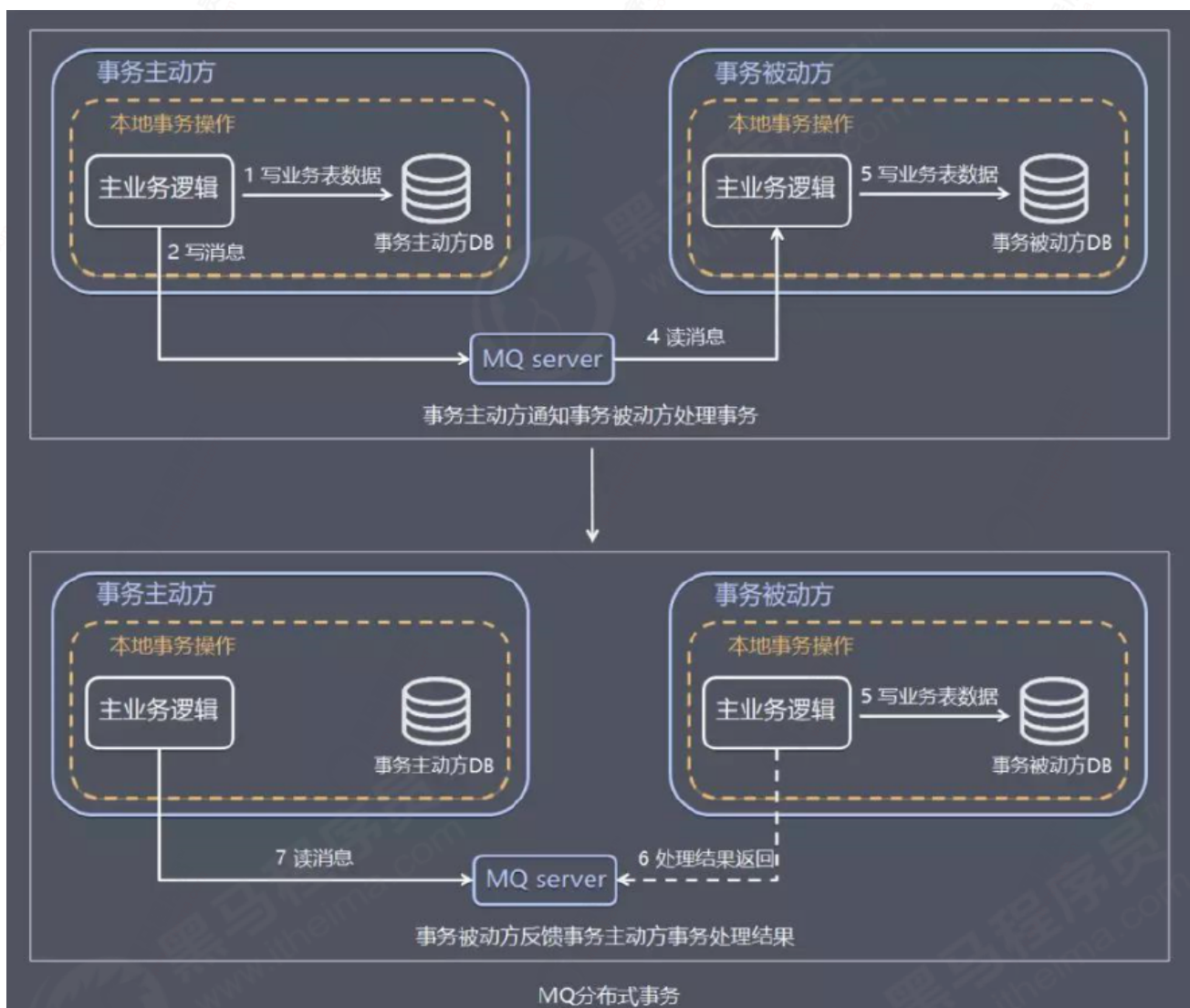
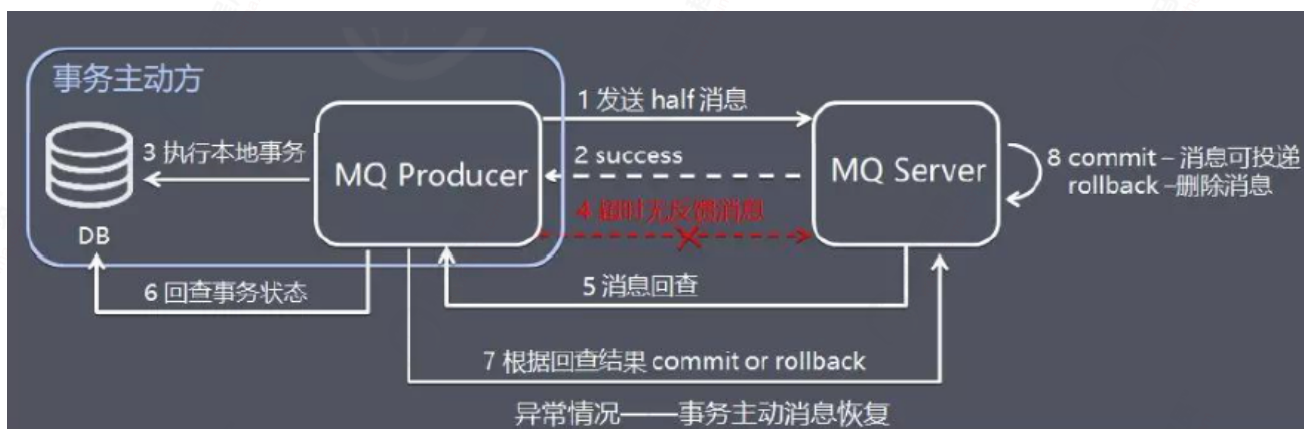
第一阶段Prepared消息，会拿到消息的地址。

第二阶段执行本地事务。

第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

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





2) 它和本地消息表的对比

分类	共同点	优势	弊端
本地消息表	都需要自己写业务补偿代码	一种非常经典的实现，避免了分布式事务，实现了最终一致性。	消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。
MQ 事务消息	都需要自己写业务补偿代码	实现了最终一致性，不需要依赖本地数据库事务。 用消息队列的方式实现分布式事务,效率较高	目前主流MQ中有ActiveMQ RocketMQ支持事务消息 实现难度较大,和业务耦合比较紧密

3 分布式事务实战-二阶段提交方案的具体应用：atomikos

3.1 atomikos

3.1.1 概述

atomikos一套开源的，JTA规范的实现。它是基于二阶段提交思想实现分布式事务的控制。是分布式刚性事务的一种解决方案。在当下互联网开发中选择此种解决方案的场景也不是很多了。

3.1.2 使用场景

通常情况下使用2pc提交方案的场景都是[单服务多数据源（多数据库）](#)的情况。在前面我们讲解分布式事务开篇时已经介绍了项目架构的演进过程，在分布式架构或微服务架构中，它面临着多个服务间的调用，这时就可能会出现其中一个服务处于ok状态，而另一个服务执行出现状况，因为二阶段提交方案中参与者不能提交事务，要等待其他参与者都ok了，才能一起提交，那么此时就会一直等待第二个服务执行返回结果，造成事务一直锁住的状态，性能不高。所以一般是单体架构配多个数据库的项目居多。

3.2 二阶段提交的案例简介

3.2.1 场景说明

一个企业级应用项目：进销存系统。系统要对针对库存记录访问日志。并且，库存系统数据库和日志数据库不是同一个数据库。

3.2.2 功能说明



← 入库的同时记录日志

3.2.3 项目简介

1) 数据库表结构

日志表		
<u>id</u>	<u>varchar(100)</u>	<u><pk></u>
操作方法名称	varchar(100)	
操作方法说明	varchar(500)	
操作人名称	varchar(100)	
操作时间	date	

库存表		
<u>库存编号</u>	<u>varchar(100)</u>	<u><pk></u>
货物编号	varchar(100)	
货物名称	varchar(200)	
仓库编号	varchar(100)	
库存数量	int	

2) 提前准备的工程

名称 ▲	类型
java代码	文件夹
测试类	文件夹
配置文件	文件夹

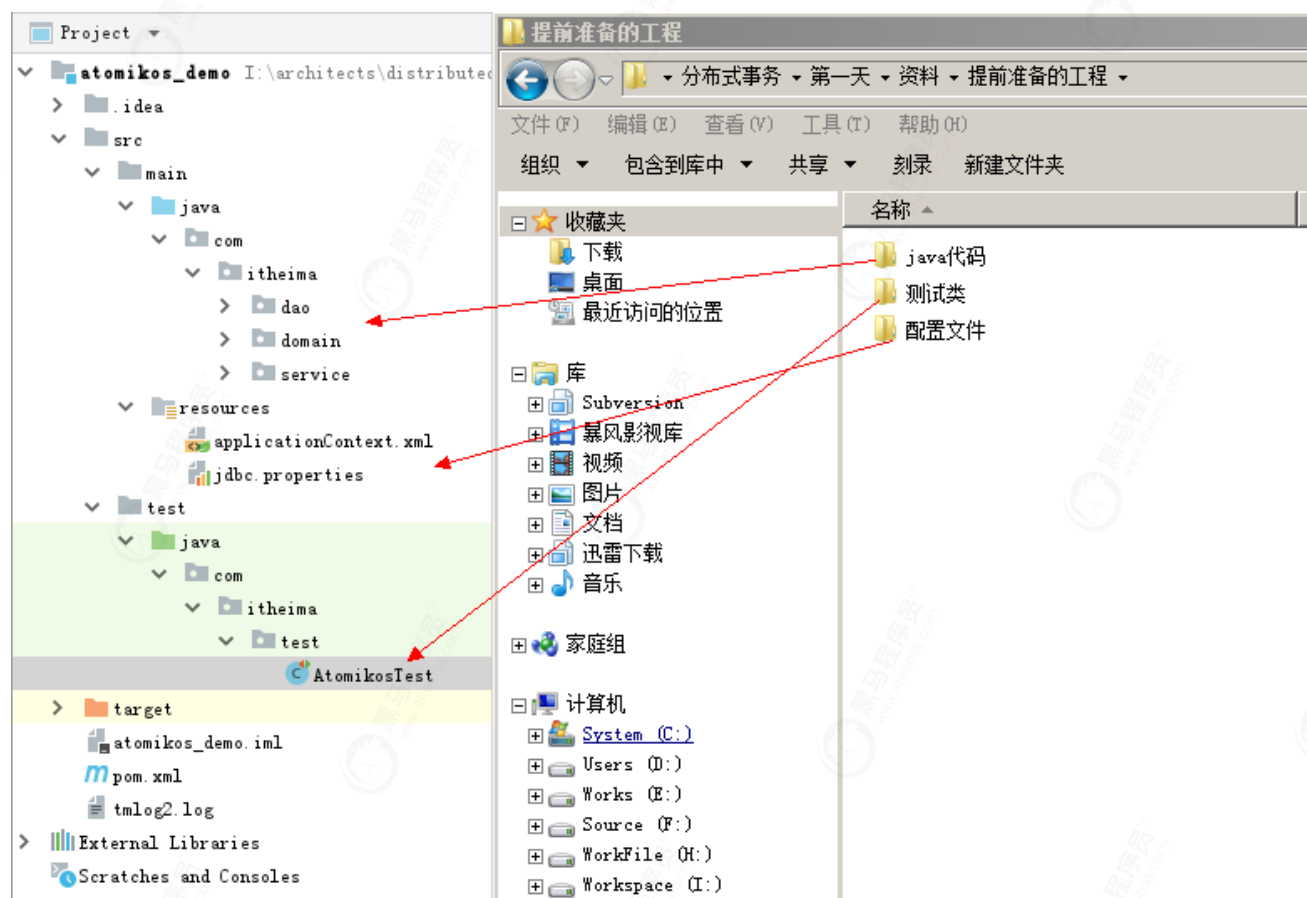
3.3 测试案例

3.3.1 导入数据库

```
CREATE DATABASE `stockdb`;
DROP TABLE IF EXISTS `t_stock`;
CREATE TABLE `t_stock` (
  `id` varchar(100) NOT NULL,
  `product_id` varchar(100) DEFAULT NULL,
  `product_name` varchar(200) DEFAULT NULL,
  `stock_id` varchar(100) DEFAULT NULL,
  `quantity` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
CREATE DATABASE `logdb`;
DROP TABLE IF EXISTS `t_log_info`;
CREATE TABLE `t_log_info` (
  `id` varchar(100) NOT NULL,
  `method` varchar(100) DEFAULT NULL,
  `action` varchar(500) DEFAULT NULL,
  `username` varchar(100) DEFAULT NULL,
  `create_time` date DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

3.3.2 导入工程



3.3.3 编写测试类和测试方法

```
package com.itheima.test;

import com.itheima.domain.Stock;
import com.itheima.service.StockService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.UUID;

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class AtomikosTest {

    @Autowired
    private StockService stockService;

    @Test
    public void testSave(){
        //1.创建对象
        Stock stock = new Stock();
        //2.填充数据
        String id = UUID.randomUUID().toString().replace("-", "").toUpperCase();
        stock.setId(id);
        stock.setProductId("1030102301111");
        stock.setProductName("华为p8");
        stock.setQuantity(100);
        //3.执行保存
        stockService.save(stock);
    }
}
```

3.3.4 编写atomikos核心代码

```

<!--
    配置事务管理器atomikos事务管理器
-->
<bean id="atomikosTransactionManager" class="com.atomikos.icatch.jta.UserTransactionManager"
init-method="init" destroy-method="close">
    <property name="forceShutdown" value="false"/>
</bean>
<!--
    本地事务管理器
-->
<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp">
    <property name="transactionTimeout" value="300000"/>
</bean>

<!--JTA事务管理器-->
<bean id="springTransactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager">
        <ref bean="atomikosTransactionManager"/>
    </property>
    <property name="userTransaction">
        <ref bean="atomikosUserTransaction"/>
    </property>
    <property name="allowCustomIsolationLevels" value="true"/>
</bean>

<!--数据源基础配置-->
<bean id="abstractXADataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean" init-
method="init" destroy-method="close" abstract="true">
    <property name="xaDataSourceClassName"
value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
    <property name="poolSize" value="10"/>
    <property name="minPoolSize" value="10"/>
    <property name="maxPoolSize" value="30"/>
    <property name="borrowConnectionTimeout" value="60"/>
    <property name="reapTimeout" value="20"/>
    <property name="maxIdleTime" value="60"/>
    <property name="maintenanceInterval" value="60"/>
    <property name="testQuery">
        <value>SELECT 1</value>
    </property>
</bean>

<!-- 数据库基本信息配置 -->
<bean id="dataSourceStock" parent="abstractXADataSource">
    <property name="uniqueResourceName">
        <value>dataSourceStock</value>
    </property>
<!--数据库驱动-->

    <property name="xaDataSourceClassName"

```

```

value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
    <property name="xaProperties">
        <props>
            <prop key="URL">${jdbc.stock.url}</prop>
            <prop key="user">${jdbc.stock.username}</prop>
            <prop key="password">${jdbc.stock.password}</prop>
        </props>
    </property>
</bean>

<!-- 日志数据源-->
<bean id="dataSourceLog" parent="abstractXADataSource">
    <property name="uniqueResourceName">
        <value>dataSourceLog</value>
    </property>
    <property name="xaDataSourceClassName"
value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
    <property name="xaProperties">
        <props>
            <prop key="URL">${jdbc.log.url}</prop>
            <prop key="user">${jdbc.log.username}</prop>
            <prop key="password">${jdbc.log.password}</prop>
        </props>
    </property>
</bean>

<!-- SqlSessionFactoryBean的配置-->
<bean id="sqlSessionFactoryBeanStock" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="typeAliasesPackage" value="com.itheima.domain" />
    <property name="dataSource" ref="dataSourceStock"/>
</bean>

<!-- 操作日志数据源的SqlSessionFactoryBean-->
<bean id="sqlSessionFactoryBeanLog" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="typeAliasesPackage" value="com.itheima.domain" />
    <property name="dataSource" ref="dataSourceLog"/>
</bean>

<!-- 包扫描, 库存Dao包扫描-->
<bean id="mapperScannerConfigurerStock"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.itheima.dao.stock" />
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactoryBeanStock" />
</bean>

<!-- 包扫描, 日志Dao包扫描-->
<bean id="mapperScannerConfigurerLog"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.itheima.dao.log" />
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactoryBeanLog" />
</bean>

```

3.4 atomikos执行原理分析及2pc总结

3.4.1 原理分析

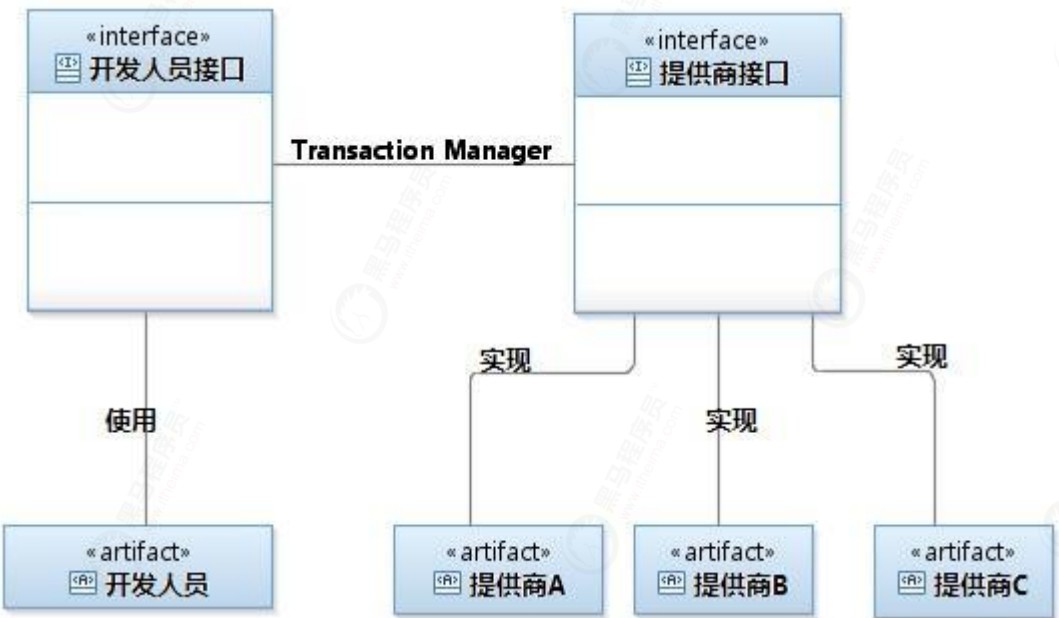
以下内容来自[IBM社区](#)

很多开发人员都会对 JTA 的内部工作机制感兴趣：我编写的代码没有任何与事务资源（如数据库连接）互动的代码，但是我的操作（数据库更新）却实实在在的被包含在了事务中，那 JTA 究竟是通过何种方式来实现这种透明性的呢？要理解 JTA 的实现原理首先需要了解其架构：它包括事务管理器（Transaction Manager）和一个或多个支持 XA 协议的资源管理器（Resource Manager）两部分，我们可以将资源管理器看做任意类型的持久化数据存储；事务管理器则承担着所有事务参与单元的协调与控制。

根据所面向对象的不同，我们可以将 JTA 的事务管理器和资源管理器理解为两个方面：面向开发人员的使用接口（事务管理器）和面向服务提供商的实现接口（资源管理器）。其中开发接口的主要部分即为上述示例中引用的 UserTransaction 对象，开发人员通过此接口在信息系统中实现分布式事务；而实现接口则用来规范提供商（如数据库连接提供商）所提供的事务服务，它约定了事务的资源管理功能，使得 JTA 可以在异构事务资源之间执行协同沟通。

以数据库为例，IBM 公司提供了实现分布式事务的数据库驱动程序，Oracle 也提供了实现分布式事务的数据库驱动程序，在同时使用 DB2 和 Oracle 两种数据库连接时，JTA 即可以根据约定的接口协调者两种事务资源从而实现分布式事务。

正是基于统一规范的不同实现使得 JTA 可以协调与控制不同数据库或者 JMS 厂商的事务资源，其架构如下图所示：



开发人员使用开发人员接口，实现应用程序对全局事务的支持；各提供商（数据库，JMS 等）依据提供商接口的规范提供事务资源管理功能；事务管理器（TransactionManager）将应用对分布式事务的使用映射到实际的事务资源并在事务资源间进行协调与控制。下面，本文将对包括 UserTransaction、Transaction 和 TransactionManager 在内的三个主要接口以及其定义的方法进行介绍。

1) 面向开发人员的 UserTransaction

开发人员通常只使用此接口实现 JTA 事务管理，其定义了如下的方法：

- **begin()**- 开始一个分布式事务，（在后台 TransactionManager 会创建一个 Transaction 事务对象并把此对象通过 ThreadLocale 关联到当前线程上）
- **commit()**- 提交事务（在后台 TransactionManager 会从当前线程下取出事务对象并把此对象所代表的事务提交）
- **rollback()**- 回滚事务（在后台 TransactionManager 会从当前线程下取出事务对象并把此对象所代表的事务回滚）
- **getStatus()**- 返回关联到当前线程的分布式事务的状态 (Status 对象里边定义了所有的事务状态，感兴趣的读者可以参考 API 文档)
- **setRollbackOnly()**- 标识关联到当前线程的分布式事务将被回滚

2) 面向提供商的 TransactionManager 和 Transaction

Transaction 代表了一个物理意义上的事务，在开发人员调用 UserTransaction.begin() 方法时 TransactionManager 会创建一个 Transaction 事务对象（标志着事务的开始）并把此对象通过 ThreadLocale 关联到当前线程。UserTransaction 接口中的 commit()、rollback()、getStatus() 等方法都将最终委托给 Transaction 类的对应方法执行。Transaction 接口定义了如下的方法：

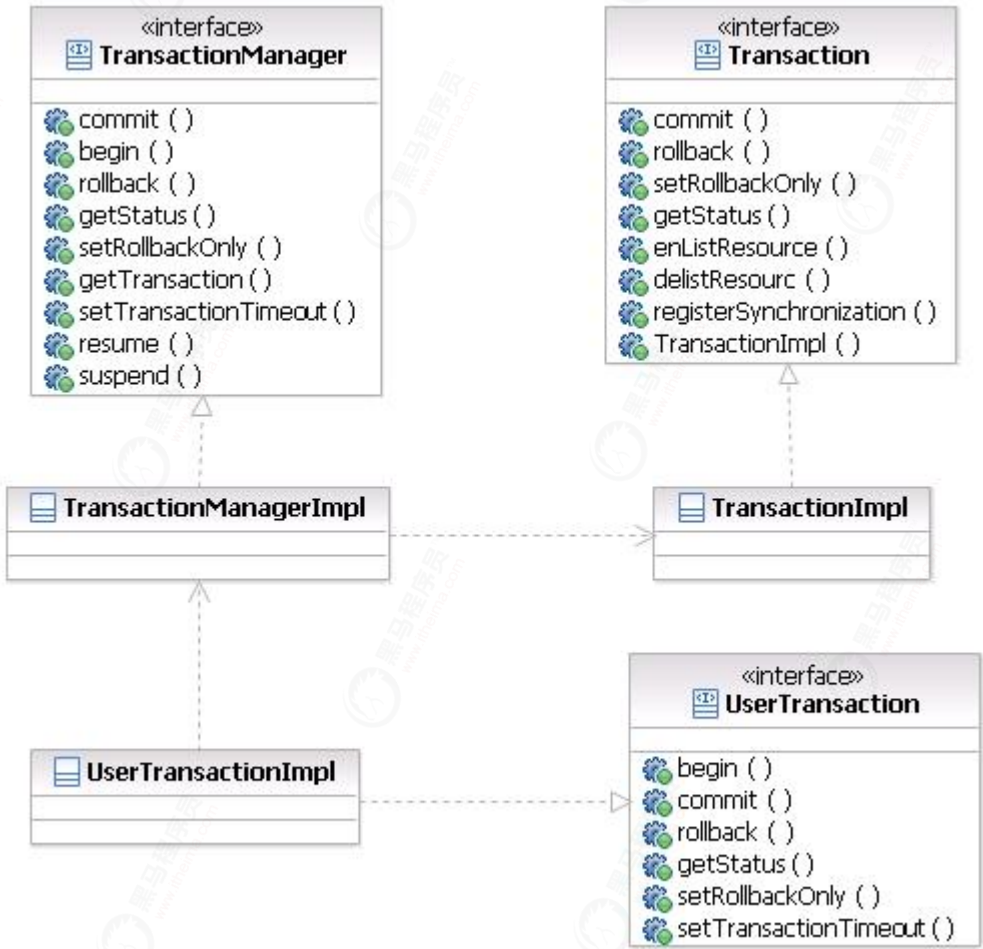
- **commit()**- 协调不同的事务资源共同完成事务的提交
- **rollback()**- 协调不同的事务资源共同完成事务的回滚
- **setRollbackOnly()**- 标识关联到当前线程的分布式事务将被回滚
- **getStatus()**- 返回关联到当前线程的分布式事务的状态
- **enListResource(XAResource xaRes, int flag)**- 将事务资源加入到当前的事务中（在上述示例中，在对数据库 A 操作时 其所代表的事务资源将被关联到当前事务中，同样，在对数据库 B 操作时其所代表的事务资源也将被关联到当前事务中）
- **delistResourc(XAResource xaRes, int flag)**- 将事务资源从当前事务中删除
- **registerSynchronization(Synchronization sync)**- 回调接口，Hibernate 等 ORM 工具都有自己的事务控制机制来保证事务，但同时它们还需要一种回调机制以便在事务完成时得到通知从而触发一些处理工作，如清除缓存等。这就涉及到了 Transaction 的回调接口 registerSynchronization。工具可以通过此接口将回调程序注入到事务中，当事务成功提交后，回调程序将被激活。

TransactionManager 本身并不承担实际的事务处理功能，它更多的是充当用户接口和实现接口之间的桥梁。下面列出了 TransactionManager 中定义的方法，可以看到此接口中的大部分事务方法与 UserTransaction 和 Transaction 相同。在开发人员调用 UserTransaction.begin() 方法时 TransactionManager 会创建一个 Transaction 事务对象（标志着事务的开始）并把此对象通过 ThreadLocale 关联到当前线程上；同样 UserTransaction.commit() 会调用 TransactionManager.commit()，方法将从当前线程下取出事务对象 Transaction 并把此对象所代表的事务提交，即调用 Transaction.commit()

- **begin()**- 开始事务
- **commit()**- 提交事务
- **rollback()**- 回滚事务
- **getStatus()**- 返回当前事务状态
- **setRollbackOnly()**
- **getTransaction()**- 返回关联到当前线程的事务
- **setTransactionTimeout(int seconds)**- 设置事务超时时间
- **resume(Transaction tobj)**- 继续当前线程关联的事务
- **suspend()**- 挂起当前线程关联的事务

在系统开发过程中会遇到需要将事务资源暂时排除的操作，此时就需要调用 suspend() 方法将当前的事务挂起：在此方法后面所做的任何操作将不会被包括在事务中，在非事务性操作完成后调用 resume() 以继续事务（注：要进行此操作需要获得 TransactionManager 对象，其获得方式在不同的 J2EE 应用服务器上是不一样的）下面将通过具体的代码向读者介绍 JTA 实现原理。

下图列出了示例实现中涉及到的 Java 类，其中 UserTransactionImpl 实现了 UserTransaction 接口，TransactionManagerImpl 实现了 TransactionManager 接口，TransactionImpl 实现了 Transaction 接口：



3.4.2 二阶段提交总结

二阶段提交作为早期分布式事务的解决方案，逐渐的淡出了主流方案的圈子。这里面其最重要的原因就是它是刚性事务，即需要满足强一致性。它的优点就是可以在多数据库间实现事务控制，而摆脱单一数据库使用事务的宿命。但是阻塞式这个缺点确是致命的，因为参与全局事务的数据库被动听从事务管理器的命令，执行或放弃事务，如果运行事务管理器的机器宕机，那整个系统就不能用了。当然，在极端情况下还可能同时影响其他系统，如果事务管理器挂了，但是这个数据库的表锁还没释放，因为数据库还在等待事务管理器的命令，因此，使用这个数据库的其他应用也会收到影响。

4 分布式事务实战-事务消息：RocketMQ

4.1 关于RocketMQ

4.1.1 RocketMQ简介

RocketMQ是阿里巴巴开发的一款原生分布式消息队列，并且其内部声称不遵循任何规范（主要是JMS），目前已经被apache软件基金会收录。它是目前实现市场上为数不多的支持事务消息的消息中间件。

通过访问[官方网址](#)，看到如下页面：

Apache RocketMQ

Apache RocketMQ™ is a unified messaging engine, lightweight data processing platform.

[Latest release v4.7.0](#)



[Getting Started](#)



通过点击[Getting Started](#)，前往详情页面，里面有下载专区，可以下载对应的RocketMQ。同时，它还给我们提供了[github](#)网址，里面有中文的资料介绍，如下图显示：

Learn it & Contact us

- Mailing Lists: <https://rocketmq.apache.org/about/contact/>
- Home: <https://rocketmq.apache.org>
- Docs: <https://rocketmq.apache.org/docs/quick-start/>
- Issues: <https://github.com/apache/rocketmq/issues>
- Rips: <https://github.com/apache/rocketmq/wiki/RocketMQ-Improvement-Proposal>
- Ask: <https://stackoverflow.com/questions/tagged/rocketmq>
- Slack: <https://rocketmq-invite-automation.herokuapp.com/>

USER GUIDE

Why RocketMQ

Quick Start

Simple Example

Order Example

Broadcasting Example

Schedule Example

Batch Example

Filter Example

Logappender Example

OpenMessaging Example

Transaction Example

FAQ

Quick Start

This quick start guide is a detailed instruction of setting up RocketMQ messaging system on your local machine to send and receive messages.

More Details:

- English : <https://github.com/apache/rocketmq/tree/master/docs/en>
- 中文 : <https://github.com/apache/rocketmq/tree/master/docs/cn>

Prerequisite

ON THIS PAGE

DDDDCCCCC

Apache RocketMQ开发者指南

这个开发者指南是帮助您快速了解,并使用 Apache RocketMQ

1. 概念和特性

- [概念\(Concept\)](#) : 介绍RocketMQ的基本概念模型。
- [特性\(Features\)](#) : 介绍RocketMQ实现的功能特性。

2. 架构设计

- [架构\(Architecture\)](#) : 介绍RocketMQ部署架构和技术架构。
- [设计\(Design\)](#) : 介绍RocketMQ关键机制的设计原理, 主要包括消息存储、通信机制、消息过滤、负载均衡、事务消息等。

3. 样例

- [样例\(Example\)](#) : 介绍RocketMQ的常见用法, 包括基本样例、顺序消息样例、延时消息样例、批量消息样例、过滤消息样例、事务消息样例等。

4. 最佳实践

- [最佳实践 \(Best Practice \)](#) : 介绍RocketMQ的最佳实践, 包括生产者、消费者、Broker以及NameServer的最佳实践, 客户端的配置方式以及JVM和linux的最佳参数配置。
- [消息轨迹指南\(Message Trace\)](#) : 介绍RocketMQ消息轨迹的使用方法。
- [权限管理\(Auth Management\)](#) : 介绍如何快速部署和使用支持权限控制特性的RocketMQ集群。
- [Dledger快速搭建\(Quick Start\)](#) : 介绍Dledger的快速搭建方法。
- [集群部署\(Cluster Deployment\)](#) : 介绍Dledger的集群部署方式。

这里面有RocketMQ的相关介绍, 我们的课程主要是针对RocketMQ的事务消息的讲解, 所以在课程中就不给同学们逐一展开说明了, 同学们可以根据自身情况前往了解。

4.1.2 下载安装

前往RocketMQ的[下载页面](#), 下载最新的版本, 如下图所示:

4.7.0 release

- Released March 16, 2020
- [Release Notes](#)
- Source: [rocketmq-all-4.7.0-source-release.zip](#) [PGP] [SHA512]
- Binary: [rocketmq-all-4.7.0-bin-release.zip](#) [PGP] [SHA512]

4.6.1 release

- Released Feb 14, 2020
- [Release Notes](#)
- Source: [rocketmq-all-4.6.1-source-release.zip](#) [PGP] [SHA512]
- Binary: [rocketmq-all-4.6.1-bin-release.zip](#) [PGP] [SHA512]

4.6.0 release

- Released Nov 25, 2019
- [Release Notes](#)
- Source: [rocketmq-all-4.6.0-source-release.zip](#) [PGP] [SHA512]
- Binary: [rocketmq-all-4.6.0-bin-release.zip](#) [PGP] [SHA512]

4.2 RocketMQ中的事务消息

4.2.1 事务消息的简介

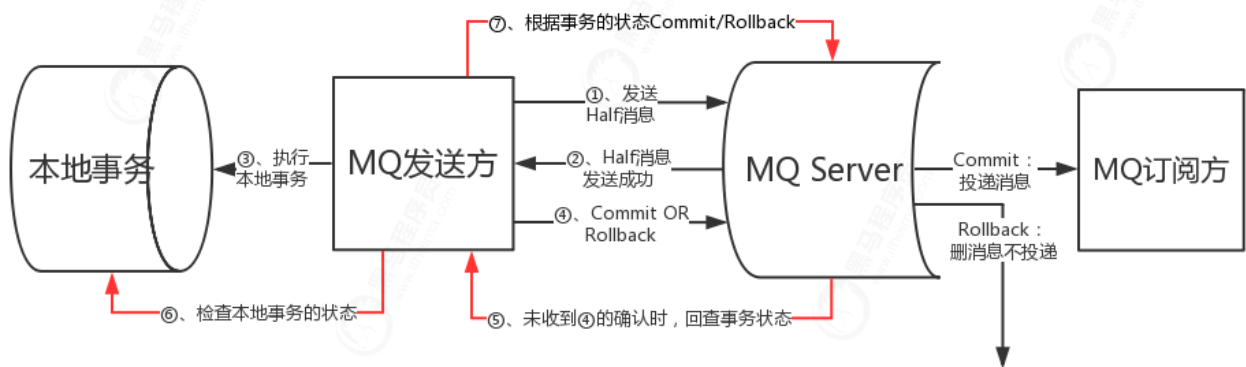
消息队列，相信同学们都很熟悉，在微服务架构中，两个服务间异步通信通常都会使用消息队列来保证执行效率。例如，用户微服务和短信微服务，在用户注册时，给用户发送手机验证码短信时，就会用到它。而如果同时需要事务的支持，那么通常会选择支持事务消息的消息队列来实现。

事务消息，它是消息队列中一种特殊的消息类型，只不过不是所有的消息队列产品都支持事务消息。目前支持事务消息的队列一个是阿里的RocketMQ（已经被apache收录），一个是ActiveMQ。今天我们课程使用的是RocketMQ。

RocketMQ事务消息（Transactional Message）是指应用本地事务和发送消息操作可以被定义到全局事务中，要么同时成功，要么同时失败。RocketMQ的事务消息提供类似 X/Open XA 的分布事务功能，通过事务消息能达到分布式事务的最终一致。

此处我们需要明确一件事，分布式事务和事务消息两者并没有关系，事务消息仅仅保证本地事务和MQ消息发送形成整体的原子性，而投递到MQ服务器后，消费者是否能一定消费成功是无法保证的。

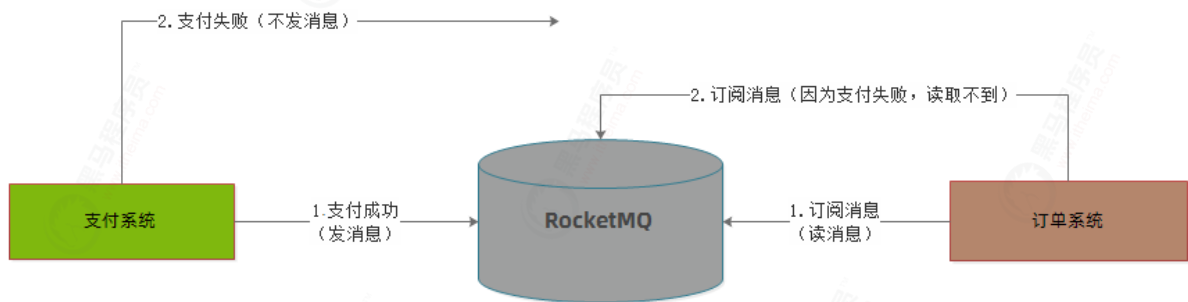
4.2.2 RocketMQ事务消息的执行流程图



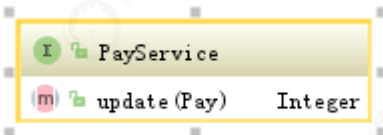
4.3 事务消息的案例简介

4.3.1 场景说明

今日课程中我们以最常见的电商项目为例，从中节选一个场景：订单和支付。我们都知道，在下单成功后，马上紧接着的就是需要付款。只有付款成功了之后，订单的状态才会改为已付款，进而继续走出库，发货，物流等等的流程，而如果订单迟迟不付款的话，超过一个时限之后就自动关闭了。下图描述了场景的业务流程：



4.3.2 功能说明



4.3.3 项目简介

1) 数据库表结构

```
DROP TABLE IF EXISTS `pay`;
CREATE TABLE `pay` (
  `id` varchar(100) NOT NULL,
  `ispay` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

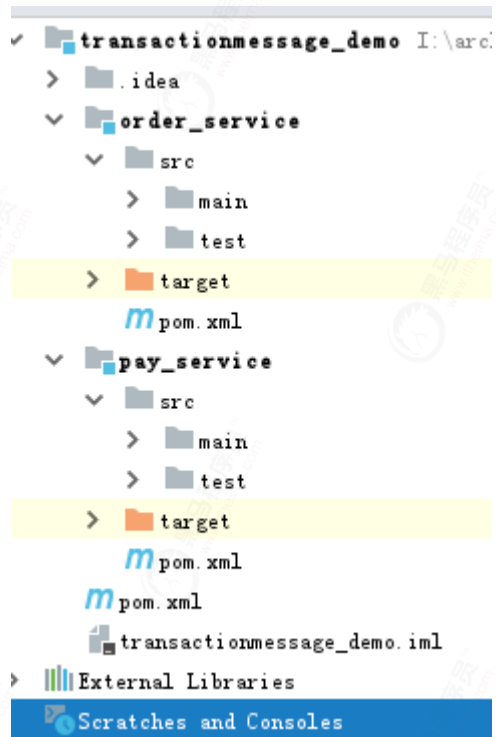
INSERT INTO `pay` VALUES ('1234', '0');
```

2) 提前准备的工程



4.4 案例实现

4.4.1 导入工程



4.4.2 导入坐标

1) 父工程坐标

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
    <relativePath/>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>

<!-- 锁定子模块使用的spring_cloud版本-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
```

```
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>spring-snapshots</id>
      <name>Spring Snapshots</name>
      <url>https://repo.spring.io/snapshot</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </pluginRepository>
    <pluginRepository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
```

2) 支付服务坐标

```

<dependencies>
    <!--mybatis 依赖-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.4</version>
    </dependency>
    <!--mybatis jar包依赖-->
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>1.3.0</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.21</version>
    </dependency>
    <!-- 整合rocketmq-->
    <dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-spring-boot-starter</artifactId>
        <version>2.0.3</version>
    </dependency>
</dependencies>

```

3) 订单服务坐标

```

<dependencies>
    <!-- https://mvnrepository.com/artifact/org.apache.rocketmq/rocketmq-client -->
    <dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-client</artifactId>
        <version>4.7.0</version>
    </dependency>
    <!-- 整合rocketmq-->
    <dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-spring-boot-starter</artifactId>
        <version>2.0.3</version>
    </dependency>
</dependencies>

```

4.4.4 编写消息生产者-支付服务的Controller

```

@RestController
public class PayController {
    @Resource
    private TransactionListener transactionListener;
    @RequestMapping(value = "/pay/updateOrder", method = RequestMethod.POST)
    public String payOrder(@RequestParam("payid") String id, @RequestParam("ispay") int ispay) {
        try {
            //创建事务消息消费者
            TransactionMQProducer transactionMQProducer = new
TransactionMQProducer("txmessage_trans-client-group");
            //指定链接的服务器地址(nameserver)
            transactionMQProducer.setNamesrvAddr("127.0.0.1:9876");
            //创建消息回查的类,我们自己的监听器
            transactionMQProducer.setTransactionListener(transactionListener);

            //创建发送的消息
            Message message = new Message(
                "txmessage_topic", "txmessage_tags", "txmessage_keys", "txmessage的消
息".getBytes(RemotingHelper.DEFAULT_CHARSET)
            );
            //启动发送者
            transactionMQProducer.start();
            //发送消息
            //传递参数is和ispay
            Map payAgrs=new HashMap();
            payAgrs.put("id", id);
            payAgrs.put("ispay", ispay);
            transactionMQProducer.sendMessageInTransaction(message, payAgrs );
            //关闭消息的发送者
            transactionMQProducer.shutdown();
        } catch (Exception e) {
            e.printStackTrace();
            return "发送消息给mq失败!";
        }
        //如果没有问题,
        return "发送消息给mq成功";
    }
}

```

4.4.5 编写消息监听者-支付服务的Listener

```

@Component
public class PayTransactionListener implements TransactionListener {

    //记录对应事务消息的执行状态 1:正在执行, 2: 执行成功, 3: 失败了
    //对于mq来说,正在事务发起方正在执行查询结果,只要未收到明确的commit或者rollback,都是未知结果unkown
    //对于mq来说,commit执行成功,才发送消息
    //对于mq来说,事务执行失败了将不再发送消息,并且将消息队列中的half消息干掉,以免再次扫描到再次回查
    //通过事务的id来辨别不同的事务

    private ConcurrentHashMap<String,Integer> transMap = new ConcurrentHashMap<String,Integer>
    ();

    // @Resource
    @Autowired
    private PayService payService;
    /**
     * 消息发送方执行自身业务操作的方法
     * @param msg 发送方发送的东西
     * @param arg 额外的参数
     * @return
     */
    public LocalTransactionState executeLocalTransaction(Message msg, Object arg) throws
    RuntimeException {
        //业务代码写这里

        String transactionId = msg.getTransactionId();
        //设置执行状态为正在执行,state=1
        transMap.put(transactionId, 1);
        //取id和ispay参数
        Map payArgs= (Map) arg;
        String id= (String) payArgs.get("id");
        Integer ispay= (Integer) payArgs.get("ispay");
        Pay pay = new Pay();
        pay.setId(id);
        pay.setIspay(ispay);
        try {
            //控制本地事务
            System.out.println("支付表更新开始");
            payService.update(pay);
            System.out.println("支付表更新成功");
            //测试用例1
            // int i=1/0;
            // 测试用例2 测试网络超时状态
            // Thread.sleep(70000);
            System.out.println("更新订单状态");
            System.out.println("订单已更新");
            //执行成功时,返回提交事务消息成功的标识
            transMap.put(transactionId, 2);
            // if(1==1){
            //     return LocalTransactionState.UNKNOW;
            // }
        }
    }
}

```

```

    }catch (Exception e){
        e.printStackTrace();
        //发生异常时, 返回回滚事务消息
        //执行成功时, 返回提交事务消息成功的标识
        transMap.put(transactionId, 3);
        System.out.println("事务执行失败, 事务执行状态为:" +
LocalTransactionState.ROLLBACK_MESSAGE);
        return LocalTransactionState.ROLLBACK_MESSAGE;
    }
    System.out.println("事务执行成功, 事务执行状态为:" + LocalTransactionState.COMMIT_MESSAGE);
    return LocalTransactionState.COMMIT_MESSAGE;
}

```

```

/**
 * 事务超时, 回查方法
 * @param msg: 携带要回查的事务ID
 * @return
 */
@Override
public LocalTransactionState checkLocalTransaction(MessageExt msg) {
    //根据transaction的id回查该事务的状态, 并返回给消息队列
    //未知状态: 查询事务状态, 但始终无结果, 或者由于网络原因发送不成功, 对mq来说都是未知状态, LocalTransactionState.UNKNOWN
    //正确提交返回LocalTransactionState.COMMIT_MESSAGE
    //事务执行失败返回LocalTransactionState.ROLLBACK_MESSAGE
    String transactionId = msg.getTransactionId();
    Integer state = transMap.get(transactionId);
    System.out.println("回查的事务id为:" + transactionId + ", 当前的状态为:" + state);

    if (state == 2){
        //执行成功, 返回commit
        System.out.println("回查结果为事务正确提交, 返回状态为:" +
LocalTransactionState.COMMIT_MESSAGE);
        return LocalTransactionState.COMMIT_MESSAGE;
    }else if (state == 3){
        //执行失败, 返回rollback
        System.out.println("回查结果为事务回滚, 返回状态为:" +
LocalTransactionState.ROLLBACK_MESSAGE);
        return LocalTransactionState.ROLLBACK_MESSAGE;
    }

    //正在执行
    System.out.println("回查正在执行, 返回状态为:" + LocalTransactionState.UNKNOWN);
    return LocalTransactionState.UNKNOWN;
}
}

```

4.4.6 编写消息消费者-订单服务的启动类

```
@SpringBootApplication
public class OrderApplication {

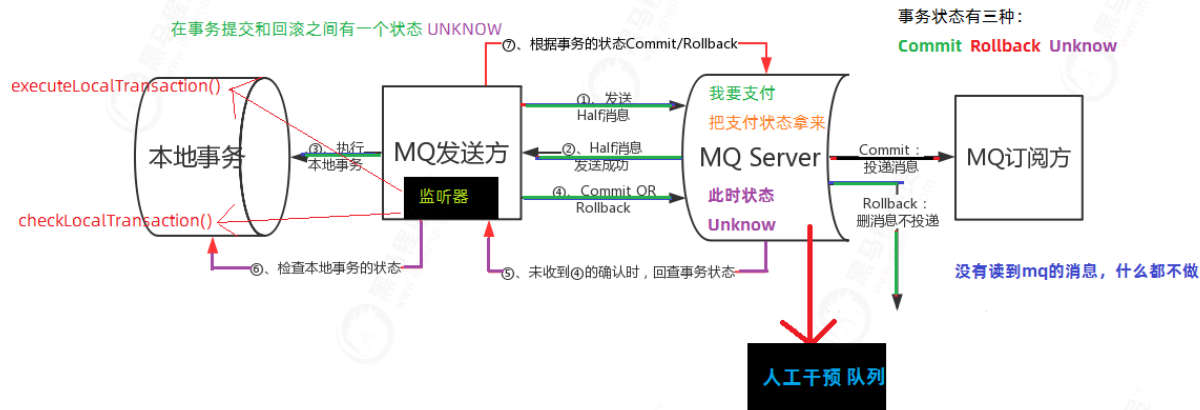
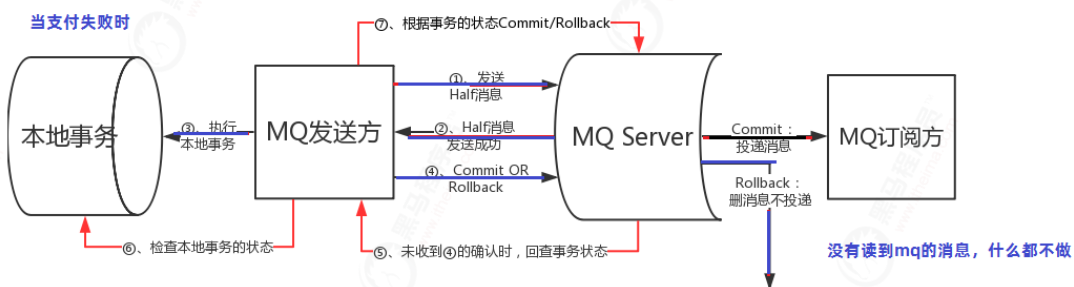
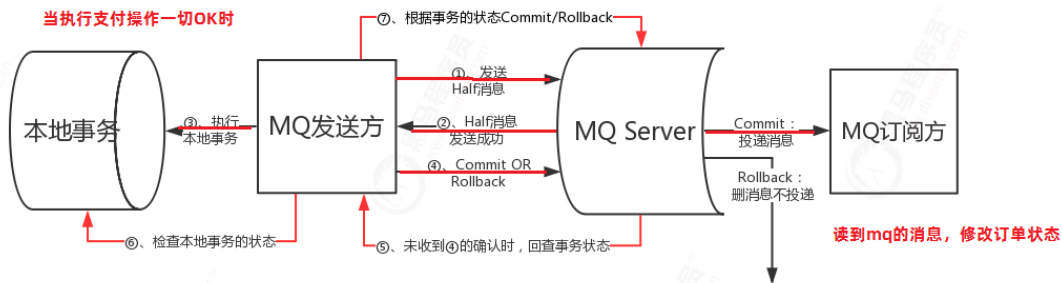
    public static void main(String[] args) throws MQClientException {
        SpringApplication.run(OrderApplication.class, args);
        //创建消息的消费者
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("txmessage_trans-client-group");
        //设置要链接的服务器地址(nameserver)
        consumer.setNamesrvAddr("127.0.0.1:9876");
        //设置单次消费的消息的数量
        consumer.setConsumeMessageBatchMaxSize(5);
        //设置消息消费的顺序
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        //设置消费者监听哪些消息
        consumer.subscribe("txmessage_topic", "txmessage_tags");
        //进行消息的接收，并返回接收消息的结果
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> list,
                ConsumeConcurrentlyContext consumeConcurrentlyContext) {

                try {
                    for(MessageExt mes :list){
                        String topic = mes.getTopic();
                        String tags = mes.getTags();
                        String keys = mes.getKeys();
                        String s = new String(mes.getBody(), "utf-8");
                        String transactionId = mes.getTransactionId();
                        System.out.println("接收到的transactionid:"+transactionId+",
                            topic:"+topic+",tags:"+tags+",消息:"+s);
                    }
                } catch (Exception e){
                    e.printStackTrace();
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        //启动消费者
        System.out.println("启动完成");
        consumer.start();
    }
}
```

4.5 执行过程分析

事务消息的发起者 (MQ发送方)
 事务消息的消费者 (MQ订阅方)
 MQ服务 (RocketMQ)
 本地事务 (MySQL)



1. 消息的生产者
2. 监听器
 - 需要做的:
 - a. 执行本地事务, 返回执行结果 (commit,rollback unknow其中之一)
 - b. 检查本地事务的执行状态 (commit,rollback unknow其中之一)