### 1、Spring Transaction Management

事务管理的角度：事务包括全局事务和本地事务。全局事务也叫分布式事务。

#### 1、什么是分布式事务

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、分布式事务的产生的原因

### 2.1、数据库分库分表

当数据库单表一年产生的数据超过1000W，那么就要考虑分库分表，具体分库分表的原理在此不做解释，以后有空详细说，简单的说就是原来的一个数据库变成了多个数据库。这时候，如果一个操作既访问01库，又访问02库，而且要保证数据的一致性，那么就要用到分布式事务。

### 2.2、应用SOA化

所谓的SOA化，就是业务的服务化。比如原来单机支撑了整个电商网站，现在对整个网站进行拆解，分离出了订单中心、用户中心、库存中心。对于订单中心，有专门的数据库存储订单信息，用户中心也有专门的数据库存储用户信息，库存中心也会有专门的数据库存储库存信息。这时候如果要同时对订单和库存进行操作，那么就会涉及到订单数据库和库存数据库，为了保证数据一致性，就需要用到分布式事务。

## 3、事务的ACID特性

### 3.1、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。

### 3.2、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。

### 3.3、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。

### 3.4、持久性（D）

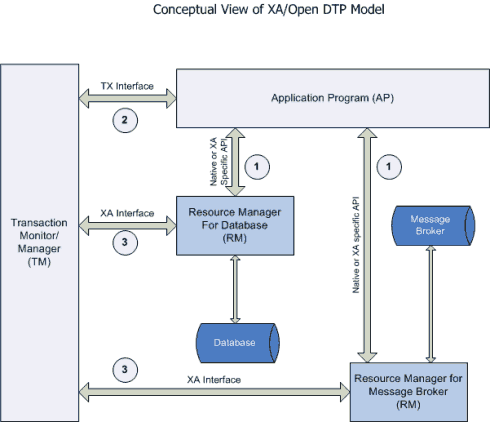
所谓的持久性，就是说一单事务完成了，那么事务对数据所做的变更就完全保存在了数据库中，即使发生停电，系统宕机也是如此。

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，  
比如用户A账户表和用户B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。

二 XA

XA是由X/Open组织提出的分布式事务的规范。XA规范主要 定义了(全局)事务管理器(Transaction Manager)和(局部)资源管理器(Resource Manager)之间的接口。  
XA接口是双向的系统接口，在事务管理器（Transaction Manager）以及一个或多个资源管理器（Resource Manager）之间形成通信桥梁。  
XA之所以需要引入事务管理器是因为，在分布式系统中，从理论上讲（参考Fischer等的论文），两台机器理论上无 法达到一致的状态，需要引入一个单点进行协调。  
事务管理器控制着全局事务，管理事务生命周期，并协调资源。资源管理器负责控制和管理实际资源（如数据库或 JMS队列）。  
下图说明了事务管理器、资源管理器，与应用程序之间的关系：



三 两阶段提交协议

分布式事务必须满足传统事务的特性，即原子性，一致性，分离性和持久性。但是分布式事务处理过程中，某些场地(Server)可能发生故障，  
或 者由于网络发生故障而无法访问到某些场地。为了防止分布式系统部分失败时产生数据的不一致性。  
在分布式事务的控制中采用了两阶段提交协议（Two- Phase Commit Protocol）。即事务的提交分为两个阶段：

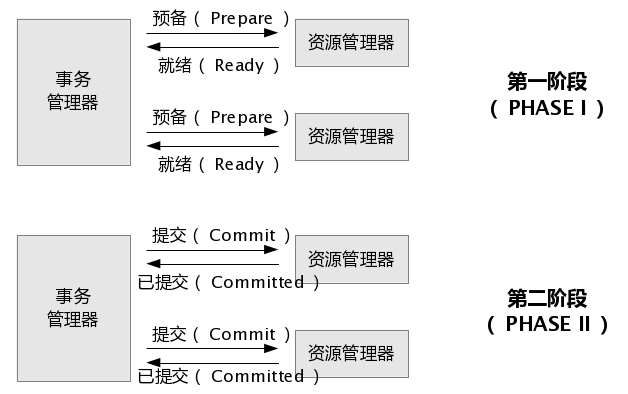
　　预提交阶段(Pre-Commit Phase)  
　　决策后阶段（Post-Decision Phase）

　　两阶段提交用来协调参与一个更新中的多个服务器的活动，以防止分布式系统部分失败时产生数据的不一致性。例如，如果一个更新操作要求位于三个不同结点上的记录被改变，且其中只要有一个结点失败，另外两个结点必须检测到这个失败并取消它们所做的改变。

　　为了支持两阶段提交，一个分布式更新事务中涉及到的服务器必须能够相互通信。一般来说一个服务器会被指定为"控制"或"提交"服务器并监控来自其它服务器的信息。

　 　在分布式更新期间，各服务器首先标志它们已经完成（但未提交）指定给它们的分布式事务的那一部分，并准备提交（以使它们的更新部分成为永久性的）。这是 两阶段提交的第一阶段。如果有一结点不能响应，那么控制服务器要指示其它结点撤消分布式事务的各个部分的影响。如果所有结点都回答准备好提交，控制服务器 则指示它们提交并等待它们的响应。等待确认信息阶段是第二阶段。  
在接收到可以提交指示后，每个服务器提交分布式事务中属于自己的那一部分，并给控制服务器 发回提交完成信息。

　　在一个分布式事务中，必须有一个场地的Server作为协调者(coordinator)，它能向 其它场地的Server发出请求，并对它们的回答作出响应，由它来控制一个分布式事务的提交或撤消。该分布式事务中涉及到的其它场地的Server称为参 与者（Participant）。



事务两阶段提交的过程如下：  
　　● 两阶段提交在应用程序向协调者发出一个提交命令时被启动。这时提交进入第一阶段，即预提交阶段。在这一阶段中：  
　　(1) 协调者准备局部（即在本地）提交并在日志中写入"预提交"日志项，并包含有该事务的所有参与者的名字。  
　 (2) 协调者询问参与者能否提交该事务。一个参与者可能由于多种原因不能提交。例如，该Server提供的约束条件（Constraints）的延迟检查不符合 限制条件时，不能提交；参与者本身的Server进程或硬件发生故障，不能提交；或者协调者访问不到某参与者（网络故障），这时协调者都认为是收到了一个 否定的回答。  
　　(3) 如果参与者能够提交，则在其本身的日志中写入"准备提交"日志项，该日志项立即写入硬盘，然后给协调者发回，已准备好提交"的回答。  
　　(4) 协调者等待所有参与者的回答，如果有参与者发回否定的回答，则协调者撤消该事务并给所有参与者发出一个"撤消该事务"的消息，结束该分布式事务，撤消该事务的所有影响。

　　● 如果所有的参与者都送回"已准备好提交"的消息，则该事务的提交进入第二阶段，即决策后提交阶段。在这一阶段中：  
　　(1) 协调者在日志中写入"提交"日志项，并立即写入硬盘。  
　　(2) 协调者向参与者发出"提交该事务"的命令。各参与者接到该命令后，在各自的日志中写入"提交"日志项，并立即写入硬盘。然后送回"已提交"的消息，释放该事务占用的资源。   
　　(3) 当所有的参与者都送回"已提交"的消息后，协调者在日志中写入"事务提交完成"日志项，释放协调者占用的资源 。这样，完成了该分布式事务的提交。

     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来快速实现。

     缺点

　　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。

**四 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，往往是给你一张小票，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，当用户A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“让用户B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）让用户B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。

4.1 如何可靠保存凭证（消息）

　　有两种方法：

4.1.1 业务与消息耦合的方式

　　用户A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）；

|  |  |
| --- | --- |
| 1  2  3  4  5 | <span style="color: #000000;">Begin transaction  update A set amount</span>=amount-10000 where userId=1<span style="color: #000000;">;  insert into message(userId, amount,status) values(</span>1, 10000, 1<span style="color: #000000;">);  End transaction  commit;</span> |

　　上述事务能保证只要用户A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用户B，用户B处理成功后发送回复成功消息，用户A收到回复后删除该条消息数据。

4.1.2 业务与消息解耦方式

　　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）用户A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）当用户A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）当用户A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用户A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步用户A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。

4.2 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，以我们用户A转账到用户B为例，如果相同的消息被重复投递两次，那么我们用户B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？比如用户B处理完消息msg后，发送了处理成功的消息给用户A，正常情况下用户A应该要删除消息msg，但如果用户A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，在用户B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）。