分布式系统事务一致性解决方案

## 开篇

在OLTP系统领域，我们在很多业务场景下都会面临事务一致性方面的需求，例如最经典的Bob给Smith转账的案例。传统的企业开发，系统往往是以单体应用形式存在的，也没有横跨多个数据库。我们通常只需借助开发平台中特有数据访问技术和框架（例如Spring、JDBC、ADO.NET），结合关系型数据库自带的事务管理机制来实现事务性的需求。关系型数据库通常具有ACID特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

而大型互联网平台往往是由一系列分布式系统构成的，开发语言平台和技术栈也相对比较杂，尤其是在SOA和微服务架构盛行的今天，一个看起来简单的功能，内部可能需要调用多个“服务”并操作多个数据库或分片来实现，情况往往会复杂很多。单一的技术手段和解决方案，已经无法应对和满足这些复杂的场景了。

## 分布式系统的特性

对分布式系统有过研究的读者，可能听说过“CAP定律”、“Base理论”等，非常巧的是，化学理论中ACID是酸、Base恰好是碱。这里笔者不对这些概念做过多的解释，有兴趣的读者可以查看相关参考资料。

在分布式系统中，同时满足“CAP定律”中的“一致性”、“可用性”和“分区容错性”三者是不可能的，这比现实中找对象需同时满足“高、富、帅”或“白、富、美”更加困难。在互联网领域的绝大多数的场景，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

## CAP定理

CAP理论认为在分布式的环境下设计和部署系统时，有3个核心的需求：Consistency，Availability和Partition Tolerance，即CAP。

* Consistency：一致性，这个和数据库ACID的一致性类似，但这里关注的所有数据节点上的数据一致性和正确性，而数据库的ACID关注的是在在一个事务内，对数据的一些约束。系统在执行过某项操作后仍然处于一致的状态。在分布式系统中，更新操作执行成功后所有的用户都应该读取到最新值。
* Availability：可用性，每一个操作总是能够在一定时间内返回结果。需要注意“一定时间”和“返回结果”。“一定时间”是指，系统结果必须在给定时间内返回。“返回结果”是指系统返回操作成功或失败的结果。
* Partition Tolerance：分区容忍性，是否可以对数据进行分区。这是考虑到性能和可伸缩性。

## 数据一致性模型

强一致性：

当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。

弱一致性：

系统并不保证续进程或者线程的访问都会返回最新的更新过的值。用户读到某一操作对系统特定数据的更新需要一段时间，我们称这段时间为“不一致性窗口”。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。

最终一致性：

是弱一致性的一种特例。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。DNS是一个典型的最终一致性系统。

## 两阶段提交协议

提到分布式系统，必然要提到分布式事务。要想理解分布式事务，不得不先介绍一下两阶段提交协议。先举个简单但不精准的例子来说明。

第一阶段，张老师作为“协调者”，给小强和小明（参与者、节点）发微信，组织他们俩明天8点在学校门口集合，一起去爬山，然后开始等待小强和小明答复。

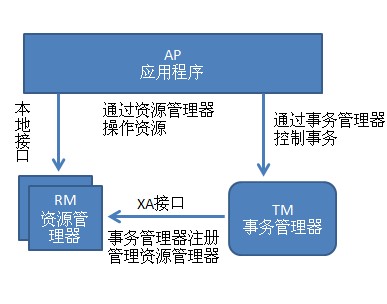
第二阶段，如果小强和小明都回答没问题，那么大家如约而至。如果小强或者小明其中一人回答说“明天没空，不行”，那么张老师会立即通知小强和小明“爬山活动取消”。

细心的读者会发现，这个过程中可能有很多问题的。如果小强没看手机，那么张老师会一直等着答复，小明可能在家里把爬山装备都准备好了却一直等着张老师确认信息。更严重的是，如果到明天8点小强还没有答复，那么就算“超时”了，那小明到底去还是不去集合爬山呢？

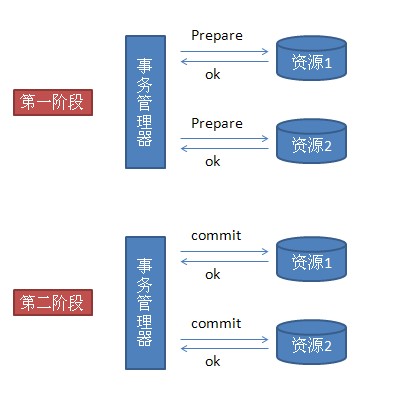
这就是两阶段提交协议的弊病，所以后来业界又引入了三阶段提交协议来解决该类问题。

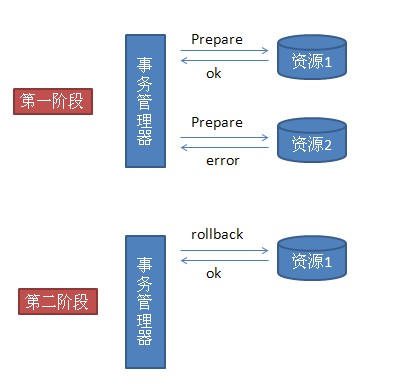
两阶段提交协议在主流开发语言平台，数据库产品中都有广泛应用和实现的。

下面来介绍一下XOpen组织提供的DTP模型图：



XA协议指的是TM（事务管理器）和RM（资源管理器）之间的接口。目前主流的关系型数据库产品都是实现了XA接口的。JTA(Java Transaction API)是符合X/Open DTP模型的，事务管理器和资源管理器之间也使用了XA协议。 本质上也是借助两阶段提交协议来实现分布式事务的，下面分别来看看XA事务成功和失败的模型图：





在JavaEE平台下，WebLogic、Webshare等主流商用的应用服务器提供了JTA的实现和支持。而在Tomcat下是没有实现的（其实笔者并不认为Tomcat能算是JavaEE应用服务器），这就需要借助第三方的框架Jotm、Automikos等来实现，两者均支持spring事务整合。

而在Windows .NET平台中，则可以借助ado.net中的TransactionScop API来编程实现，还必须配置和借助Windows操作系统中的MSDTC服务。如果你的数据库使用的mysql，并且mysql是部署在Linux平台上的，那么是无法支持分布式事务的。 由于篇幅关系，这里不展开，感兴趣的读者可以自行查阅相关资料并实践。

总结：这种方式实现难度不算太高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况。但分布式事务对性能的影响会比较大，不适合高并发和高性能要求的场景。

在服务化架构中，功能X，需要去协调后端的A、B甚至更多的原子服务。那么问题来了，假如A和B其中一个调用失败了，那可怎么办呢？

在笔者的工作中经常遇到这类问题，往往提供了一个BFF层来协调调用A、B服务。如果有些是需要同步返回结果的，我会尽量按照“串行”的方式去调用。如果调用A失败，则不会盲目去调用B。如果调用A成功，而调用B失败，会尝试去回滚刚刚对A的调用操作。

当然，有些时候我们不必严格提供单独对应的回滚接口，可以通过传递参数巧妙的实现。

这样的情况，我们会尽量把可提供回滚接口的服务放在前面。举个例子说明：

我们的某个论坛网站，每天登录成功后会奖励用户5个积分，但是积分和用户又是两套独立的子系统服务，对应不同的DB，这控制起来就比较麻烦了。解决思路：

* 把登录和加积分的服务调用放在BFF层一个本地方法中。
* 当用户请求登录接口时，先执行加积分操作，加分成功后再执行登录操作
* 如果登录成功，那当然最好了，积分也加成功了。如果登录失败，则调用加积分对应的回滚接口（执行减积分的操作）。

总结：这种方式缺点比较多，通常在复杂场景下是不推荐使用的，除非是非常简单的场景，非常容易提供回滚，而且依赖的服务也非常少的情况。

这种实现方式会造成代码量庞大，耦合性高。而且非常有局限性，因为有很多的业务是无法很简单的实现回滚的，如果串行的服务很多，回滚的成本实在太高。

## 本地消息表

这种实现方式的思路，其实是源于ebay，后来通过支付宝等公司的布道，在业内广泛使用。其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。如果不考虑性能及设计优雅，借助关系型数据库中的表即可实现。

举个经典的跨行转账的例子来描述。

第一步扣款1W，通过本地事务保证了凭证消息插入到消息表中；

第二步，通知对方银行账户上加1W了。那问题来了，如何通知到对方呢？

通常采用两种方式：

* 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。

两种方式其实各有利弊，仅仅依靠MQ，可能会出现通知失败的问题。而过于频繁的定时轮询，效率也不是最佳的（90%是无用功）。所以，我们一般会把两种方式结合起来使用。

解决了通知的问题，又有新的问题了。万一这消息有重复被消费，往用户帐号上多加了钱，那岂不是后果很严重？

仔细思考，其实我们可以消息消费方，也通过一个“消费状态表”来记录消费状态。在执行“加款”操作之前，检测下该消息（提供标识）是否已经消费过，消费完成后，通过本地事务控制来更新这个“消费状态表”。这样子就避免重复消费的问题。

总结：上诉的方式是一种非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。但是，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。所以，在真正的高并发场景下，该方案也会有瓶颈和限制的。

## MQ（非事务消息）

通常情况下，在使用非事务消息支持的MQ产品时，我们很难将业务操作与对MQ的操作放在一个本地事务域中管理。通俗点描述，还是以上述提到的“跨行转账”为例，我们很难保证在扣款完成之后对MQ投递消息的操作就一定能成功。这样一致性似乎很难保证。

先从消息生产者这端来分析，请看代码流程：

* 执行业务操作，操作数据库；
* 向MQ发送消息

根据上述流程，我们来分析下可能的情况：

* 操作数据库成功，向MQ中投递消息也成功，皆大欢喜；
* 操作数据库失败，不会向MQ中投递消息了；
* 操作数据库成功，但是向MQ中投递消息时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚。

从上面分析的几种情况来看，貌似问题都不大的。那么我们来分析下消费者端面临的问题：

* 消息出列后，消费者对应的业务操作执行成功，消费者执行MQ的ack操作。如果业务执行失败，消费者不执行MQ的ack操作。
* 尽量避免消息重复消费。如果重复消费，也不能因此影响业务结果。

这样看来消费者端也没啥大的问题，但是在生产者发送消息后，到消费者消费消息的过程中，可能存在消息丢失的情况，如果发生消息丢失，就可能导致数据的不一致。

总结：这种方式比较常见，性能和吞吐量是优于使用关系型数据库消息表的方案。如果MQ自身和业务都具有高可用性，理论上是可以满足大部分的业务场景的。而且这种方式需要容忍极少情况下数据的不一致问题。

## MQ（事务消息）

举个例子，Bob向Smith转账，那我们到底是先发送消息，还是先执行扣款操作？

好像都可能会出问题。如果先发消息，扣款操作失败，那么Smith的账户里面会多出一笔钱。反过来，如果先执行扣款操作，后发送消息，那有可能扣款成功了但是消息没发出去，Smith收不到钱。除了上面介绍的通过异常捕获和回滚的方式外，还有没有其他的思路呢？

下面以阿里巴巴的RocketMQ中间件为例，分析下其设计和实现思路。

RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事物，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。细心的读者可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，这时候发现了Prepared消息，它会向消息发送者确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？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。

总结：据笔者的了解，各大知名的电商平台和互联网公司，几乎都是采用类似的设计思路来实现“最终一致性”的。这种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大。目前主流的开源MQ（ActiveMQ、RabbitMQ、Kafka）均未实现对事务消息的支持，所以需二次开发或者新造轮子。比较遗憾的是，RocketMQ事务消息部分的代码也并未开源，需要自己去实现。

## TCC

下面是github上的tcc分布式事务框架：tcc-transaction，有兴趣可以去看看，地址为：

<https://github.com/changmingxie/tcc-transaction>

## 小结

其实分布式系统的事务一致性本身是一个技术难题，目前没有一种很简单很完美的方案能够应对所有场景。具体还是要使用者根据不同的业务场景去抉择。在某些特殊的情况下，还会有“人工补偿”的，这也是最后一道屏障。

## 附录

http://www.infoq.com/cn/articles/solution-of-distributed-system-transaction-consistency#anch143901