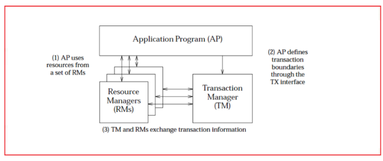
分布式服务

分布式事务解决方案

二阶段提交

首先我们来简要看下分布式事务处理的XA规范：



可知XA规范中分布式事务有AP，RM，TM组成：

其中应用程序(Application Program，简称AP)：AP定义事务边界（定义事务开始和结束）并访问事务边界内的资源。

资源管理器(Resource Manager，简称RM)：Rm管理计算机共享的资源，许多软件都可以去访问这些资源，资源包含比如数据库、文件系统、打印机服务器等。

事务管理器(Transaction Manager，简称TM)：负责管理全局事务，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、失败恢复等。

二阶段协议：

第一阶段TM要求所有的RM准备提交对应的事务分支，询问RM是否有能力保证成功的提交事务分支，RM根据自己的情况，如果判断自己进行的工作可以被提交，那就对工作内容进行持久化，并给TM回执OK；否者给TM的回执NO。RM在发送了否定答复并回滚了已经的工作后，就可以丢弃这个事务分支信息了。

第二阶段TM根据阶段1各个RM prepare的结果，决定是提交还是回滚事务。如果所有的RM都prepare成功，那么TM通知所有的RM进行提交；如果有RM prepare回执NO的话，则TM通知所有RM回滚自己的事务分支。（反向补偿）

也就是TM与RM之间是通过两阶段提交协议进行交互的.

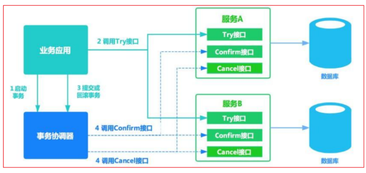
优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景。

tcc

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

- 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。



例如：A要向 B 转账，思路大概是：

```
我们有一个本地方法，里面依次调用
1、首先在 Try 阶段，要先调用远程接口把 B和 A的钱给冻结起来。
2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。
```

优点： 相比两阶段提交，可用性比较强

缺点： 数据的一致性要差一些。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

如果要用 TCC 分布式事务的话：首先需要选择某种 TCC 分布式事务框架，各个服务里就会有这个 TCC 分布式事务框架在运行，然后你原本的一个接口，要改造为 3 个逻辑，Try-Confirm-Cancel

- 先是服务调用链路依次执行 Try 逻辑
- 如果都正常的话，TCC 分布式事务框架推进执行 Confirm 逻辑，完成整个事务
- 如果某个服务的 Try 逻辑有问题，TCC 分布式事务框架感知到之后就会推进执行各个服务的 Cancel 逻辑，撤销之前执行的各种操作

这就是所谓的 TCC 分布式事务

TCC 分布式事务的核心思想，就是当遇到下面这些情况时

- 某个服务的数据库宕机了
- 某个服务自己挂了
- 部个服务的 redis，elasticsearch，MQ 等基础设施故障了
- 某些资源不足了，比如说库存不够这些

先来 Try 一下，不要把业务逻辑完成，先试试看，看各个服务能不能基本正常运转，能不能冻结我需要的资源。

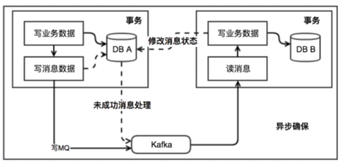
如果 Try 都 ok，也就是说，底层的数据库，redis，elasticsearch，MQ 都是可以写入数据的，并且你保留好了需要使用的一些资源（比如冻结了一部分库存）

接着，再执行各个服务的 Confirm 逻辑，基本上 Confirm 就可以很大概率保证一个分布式事务的完成了

那如果 Try 阶段某个服务失败了，比如说底层的数据库挂了，或者 redis 挂了，等等。此时就自动执行各个服务的 Cancel 逻辑，把之前的 Try 逻辑都回滚。所有服务都不要执行任何设计的业务逻辑，保证大家要么一起成功，要么一起失败。

消息异步确保

消息最终一致性应该是业界使用最多的，其核心思想是将分布式事务拆分成本地事务进行处理，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



基本思路就是：

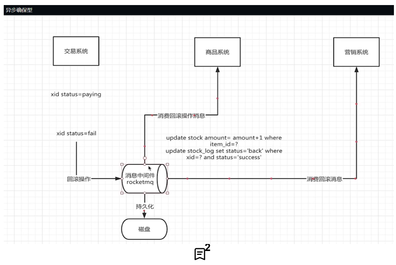
消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们是在一个数据库里面。然后消息会经过MQ发送到消费的消费方。如果消息发送失败，会进行重试发送。

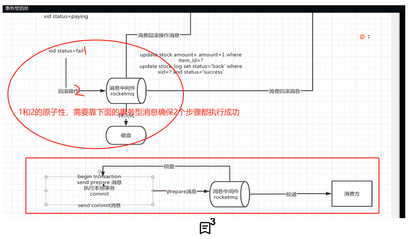
消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功了，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

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。





事务型消息

分支主题2

分支主题3

备注：

1. CSDN:https://blog.csdn.net/vincent_wen0766/article/details/114089317
2. 慕课网 分布式事务方案：异步确保型https://coding.imooc.com/lesson/480.html#mid=40615
3. 慕课网 分布式事务方案：事务型消息 https://coding.imooc.com/lesson/480.html#mid=40616
4. CSDN:https://blog.csdn.net/qq_41824825/article/details/123413868