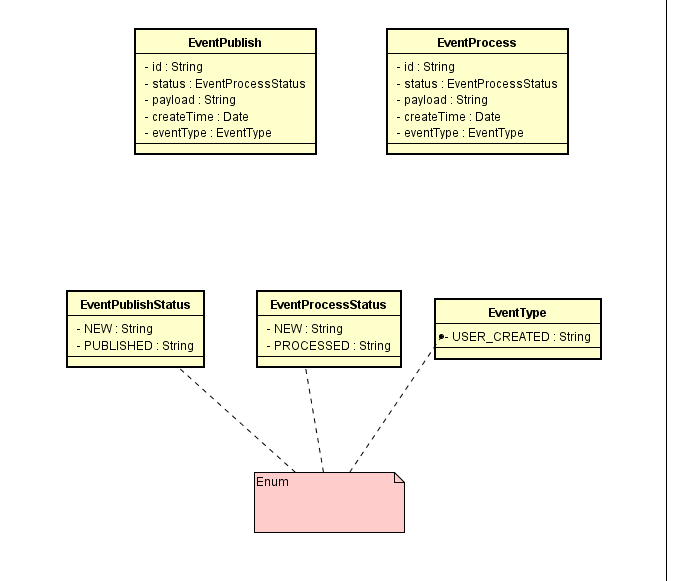
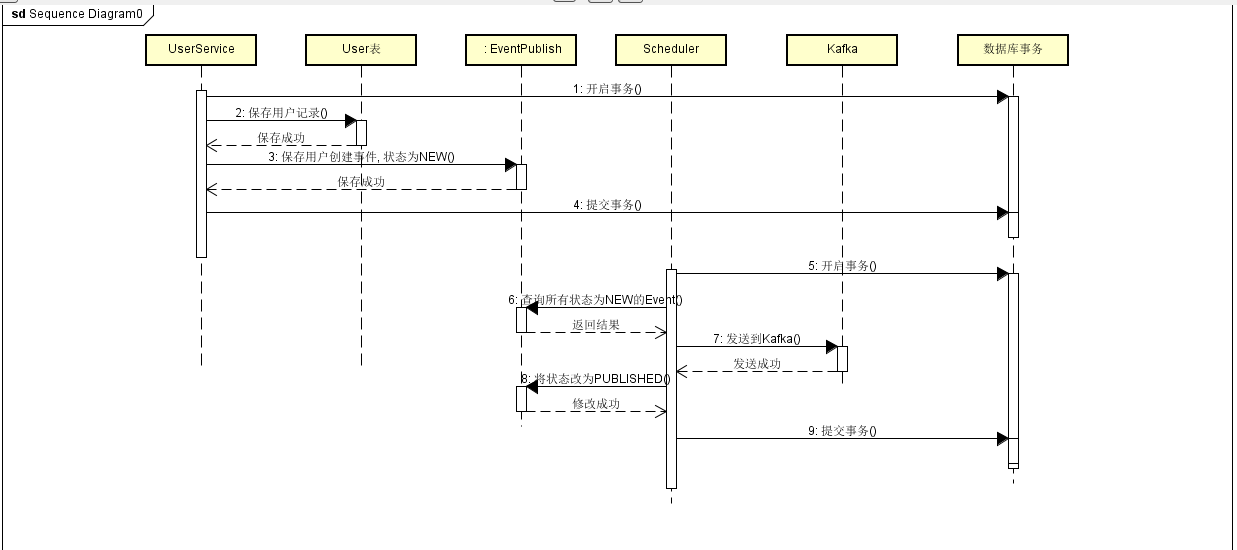
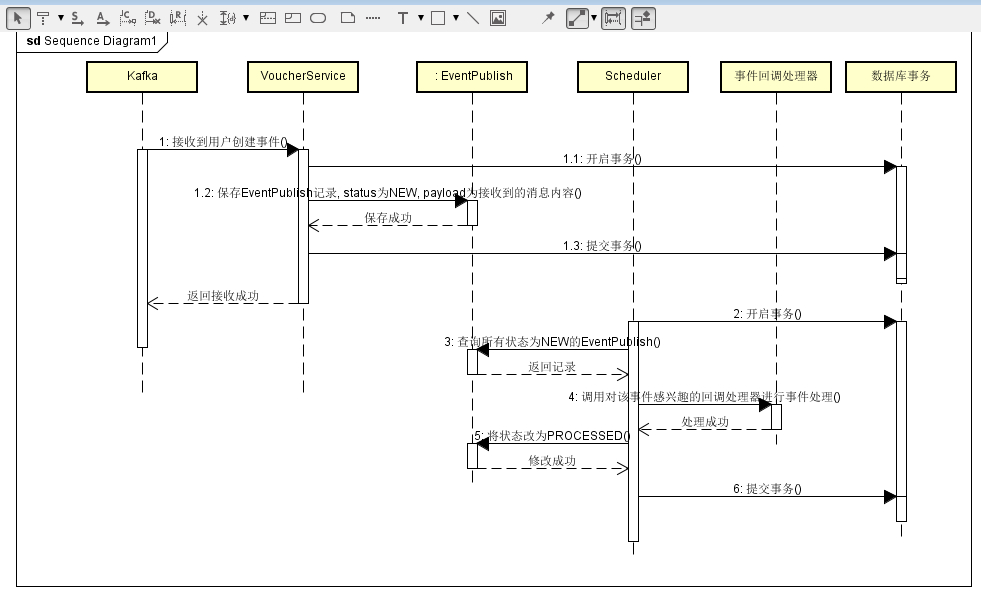
不同于单一架构应用(Monolith), 分布式环境下, 进行事务操作将变得困难, 因为分布式环境通常会有多个数据源, 只用本地数据库事务难以保证多个数据源数据的一致性. 这种情况下, 可以使用两阶段或者三阶段提交协议来完成分布式事务.但是使用这种方式一般来说性能较差, 因为事务管理器需要在多个数据源之间进行多次等待. 有一种方法同样可以解决分布式事务问题, 并且性能较好, 这就是我这篇文章要介绍的使用事件,本地事务以及消息队列来实现分布式事务.

我们从一个简单的实例入手. 基本所有互联网应用都会有用户注册的功能. 在这个例子中, 我们对于用户注册有两步操作:   
1. 注册成功, 保存用户信息.  
2. 需要给用户发放一张代金券, 目的是鼓励用户进行消费.  
如果是一个单一架构应用, 实现这个功能非常简单: 在一个本地事务里, 往用户表插一条记录, 并且在代金券表里插一条记录, 提交事务就完成了. 但是如果我们的应用是用微服务实现的, 可能用户和代金券是两个独立的服务, 他们有各自的应用和数据库, 那么就没有办法简单的使用本地事务来保证操作的原子性了. 现在来看看如何使用事件机制和消息队列来实现这个需求.(我在这里使用的消息队列是kafka, 原理同样适用于ActiveMQ/RabbitMQ等其他队列)

我们会为用户注册这个操作创建一个事件, 该事件就叫做用户创建事件(USER\_CREATED). 用户服务成功保存用户记录后, 会发送用户创建事件到消息队列, 代金券服务会监听用户创建事件, 一旦接收到该事件, 代金券服务就会在自己的数据库中为该用户创建一张代金券. 好了, 这些步骤看起来都相当的简单直观, 但是怎么保证事务的原子性呢? 考虑下面这两个场景:  
1. 用户服务在保存用户记录, 还没来得及向消息队列发送消息之前就宕机了. 怎么保证用户创建事件一定发送到消息队列了?  
2. 代金券服务接收到用户创建事件, 还没来得及处理事件就宕机了. 重新启动之后如何消费之前的用户创建事件?  
这两个问题的本质是: 如何让操作数据库和操作消息队列这两个操作成为一个原子操作. 不考虑2PC, 这里我们可以通过事件表来解决这个问题. 下面是类图. 

EventPublish是记录待发布事件的表. 其中:  
id: 每个事件在创建的时候都会生成一个全局唯一ID, 例如UUID.  
status: 事件状态, 枚举类型. 现在只有两个状态: 待发布(NEW), 已发布(PUBLISHED).  
payload: 事件内容. 这里我们会将事件内容转成json存到这个字段里.  
eventType: 事件类型, 枚举类型. 每个事件都会有一个类型, 比如我们之前提到的创建用户USER\_CREATED就是一个事件类型.  
EventProcess是用来记录待处理的事件. 字段与EventPublish基本相同.

我们首先看看事件的发布过程. 下面是用户服务发布用户创建事件的顺序图.   
1. 用户服务在接收到用户请求后开启事务, 在用户表创建一条用户记录, 并且在EventPublish表创建一条status为NEW的记录, payload记录的是事件内容, 提交事务.  
2. 用户服务中的定时器首先开启事务, 然后查询EventPublish是否有status为NEW的记录, 查询到记录之后, 拿到payload信息, 将消息发布到kafka中对应的topic.  
发送成功之后, 修改数据库中EventPublish的status为PUBLISHED, 提交事务.

下面是代金券服务处理用户创建事件的顺序图.   
1. 代金券服务接收到kafka传来的用户创建事件(实际上是代金券服务主动拉取的消息, 先忽略消息队列的实现), 在EventProcess表创建一条status为NEW的记录, payload记录的是事件内容, 如果保存成功, 向kafka返回接收成功的消息.  
2. 代金券服务中的定时器首先开启事务, 然后查询EventProcess是否有status为NEW的记录, 查询到记录之后, 拿到payload信息, 交给事件回调处理器处理, 这里是直接创建代金券记录. 处理成功之后修改数据库中EventProcess的status为PROCESSED, 最后提交事务.

回过头来看我们之前提出的两个问题:  
1. 用户服务在保存用户记录, 还没来得及向消息队列发送消息之前就宕机了. 怎么保证用户创建事件一定发送到消息队列了?  
根据事件发布的顺序图, 我们把创建事件和发布事件分成了两步操作. 如果事件创建成功, 但是在发布的时候宕机了. 启动之后定时器会重新对之前没有发布成功的事件进行发布. 如果事件在创建的时候就宕机了, 因为事件创建和业务操作在一个数据库事务里, 所以对应的业务操作也失败了, 数据库状态的一致性得到了保证.  
2. 代金券服务接收到用户创建事件, 还没来得及处理事件就宕机了. 重新启动之后如何消费之前的用户创建事件?  
根据事件处理的顺序图, 我们把接收事件和处理事件分成了两步操作. 如果事件接收成功, 但是在处理的时候宕机了. 启动之后定时器会重新对之前没有处理成功的事件进行处理. 如果事件在接收的时候就宕机了, kafka会重新将事件发送给对应服务.

通过这种方式, 我们不用2PC, 也保证了多个数据源之间状态的最终一致性.  
和2PC/3PC这种同步事务处理的方式相比, 这种异步事务处理方式具有异步系统通常都有的优点:  
1. 事务吞吐量大. 因为不需要等待其他数据源响应.  
2. 容错性好. A服务在发布事件的时候, B服务甚至可以不在线.  
缺点:  
1. 编程与调试较复杂.  
2. 容易出现较多的中间状态. 比如上面的例子, 在用户服务已经保存了用户并发布了事件, 但是代金券服务还没来得及处理之前, 用户如果登录系统, 会发现自己是没有代金券的. 这种情况可能在有些业务中是能够容忍的, 但是有些业务却不行. 所以开发之前要考虑好.

另外, 上面的流程在实现的过程中还有一些可以改进的地方:  
1. 定时器在更新EventPublish状态为PUBLISHED的时候, 可以一次批量更新多个EventProcess的状态.  
2. 定时器查询EventProcess并交给事件回调处理器处理的时候, 可以使用线程池异步处理, 加快EventProcess处理周期.  
3. 在保存EventPublish和EventProcess的时候同时保存到Redis, 之后的操作可以对Redis中的数据进行, 但是要小心处理缓存和数据库可能状态不一致问题.  
4. 针对Kafka, 因为Kafka的特点是可能重发消息, 所以在接收事件并且保存到EventProcess的时候可能报主键冲突的错误(因为重复消息id是相同的), 这个时候可以直接丢弃该消息.