1 知识回顾

1.1 目前秒杀模型

1.2 问题一 事物问题

从秒杀下单业务逻辑来讲，秒杀下单要么成功，要么失败，不存在中间状态, 从而对我们实现的秒杀服务提出了要求:

- 扣减库存 和 秒杀商品的订单以及订单邮寄信息的保存 要么全部成功，要么全部失败
- 所以，扣减库存 和 秒杀商品的订单以及订单邮寄信息的保存 的实现，应该实现为一个事物

但是，回顾一下我们秒杀下单的实现代码，我们能够实现，让秒杀服务对数据库的访问(扣减秒杀商品库存)，以及订单服务对数据库的访问(生成订单及订单的邮寄信息)，成为一个数据库事物吗?

2 传统事务回顾

2.1 事务定义

事务定义:

是数据库操作的最小工作单元，是作为单个逻辑工作单元执行的一系列操作；这些操作作为一个整体一起向系统提交，要么都执行、要么都不执行

2.2 传统事务知识点

2.2.1 四个特性（ACID）

- **原子性**：事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做
- **一致性**：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是 不一致的状态。
- **隔离性**：一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。
- **持久性**：一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

2.2.2 事务隔离级别

- 读未提交 (Read Uncommitted)：允许脏读，也就是可能读取到其他会话中未提交事务修改的数据
- 读已提交(Read Committed)：只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别(不重复读)
- 可重复读(Repeated Read)：在同一个事务内的查询都是事务开始时刻一致的，InnoDB默认级别。在SQL标准中，该隔离级别消除了不可重复读，但是还存在幻象读，但是innoDB解决了幻读
- 序列化：(Serializable)：完全串行化的读，每次读都需要获得表级共享锁，读写相互都会阻塞

2.2.3 事务的传播行为

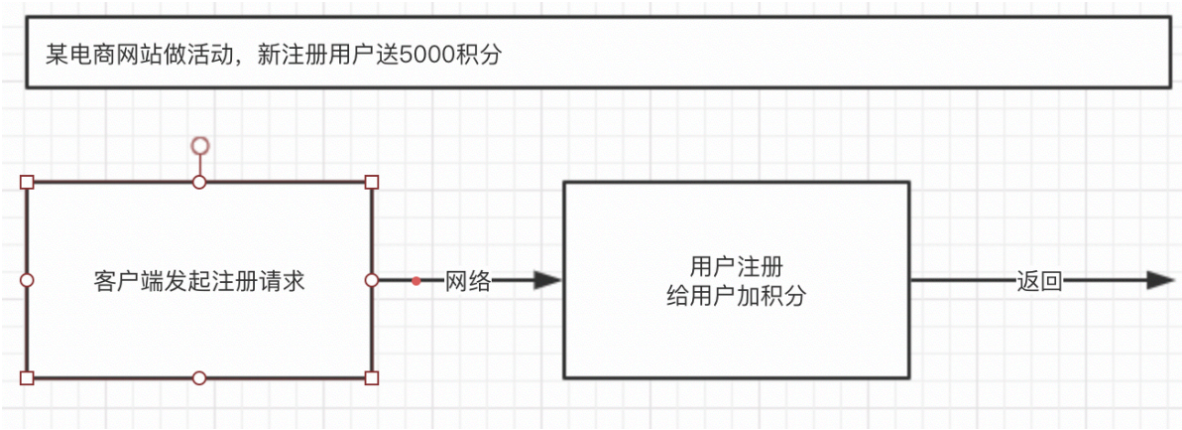
Spring 支持 7 种事务传播行为：

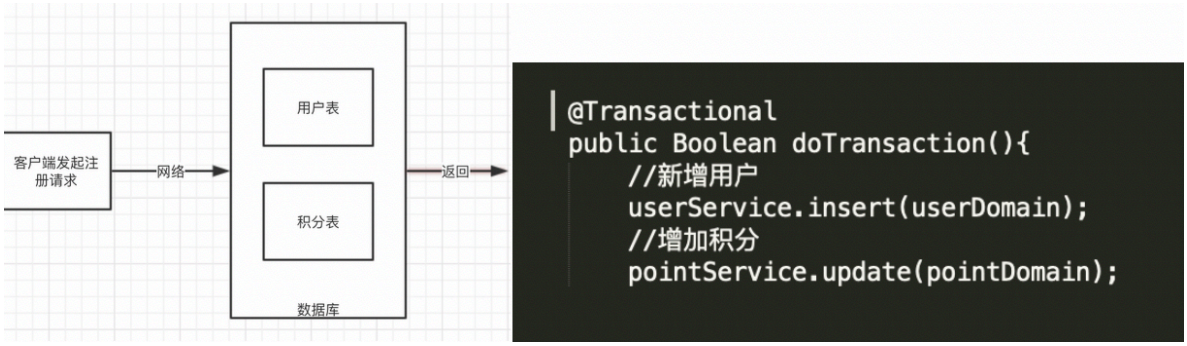
org.springframework.transaction.annotation. Propagation

```
/**
 * Enumeration that represents transaction propagation behaviors for use
 * with the {@link Transactional} annotation, corresponding to the
 * {@link TransactionDefinition} interface.
 *
 * @author Colin Sampaleanu
 * @author Juergen Hoeller
 * @since 1.2
 */
public enum Propagation {
```

Propagation.REQUIRED 是常用的事务传播行为，如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。其它传播行为大家可另查阅。

2.2.4 传统事务引用场景举例

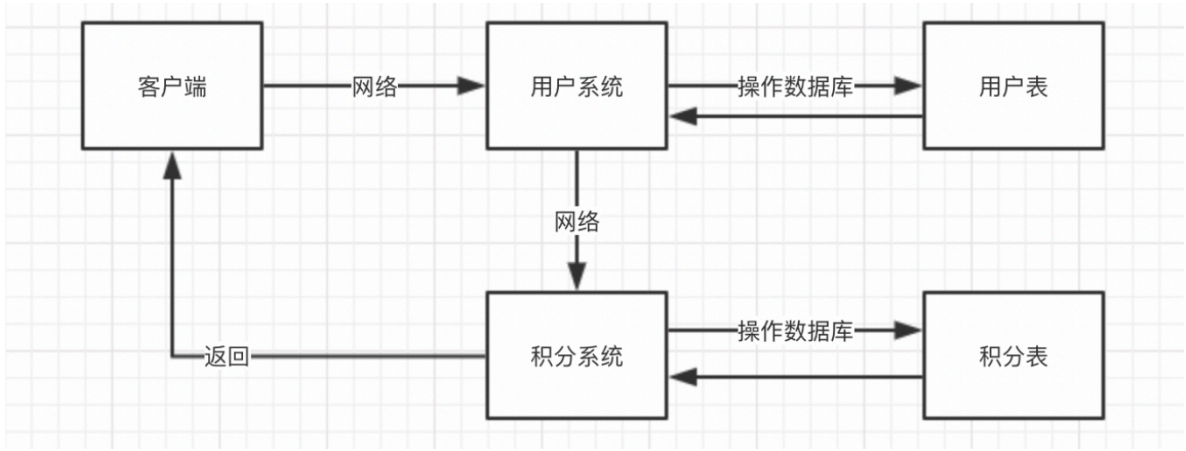




传统事务控制基础：必须得保证是同一个连接，通过jdbc操作数据源的话保证同一个connection 对象

2.3 传统事务问题

在分布式架构下，随着业务量的扩大，我们对业务进行拆分，数据库也会相应的进行分库分片，因为有着网络的不确定性，那么我们分布式环境下应该如何保证事务的ACID?



当单个数据库的性能产生瓶颈的时候，我们需要对数据库分库或者是分区，那么这个时候数据库就处于不同的服务器上了，因此基于单个数据库所控制的传统型事务已经不能在适应这种情况了，故我们需要使用分布式事务来管理这种情况

3 分布式事务

3.1 理论基础-CAP理论

3.1.1 概念

1998年，加州大学的计算机科学家 Eric Brewer 提出，分布式系统有三个指标。

- 一致性：Consistency

  集群中各个结点的数据总是一致的，因此你可以向任意结点读写数据，并总是能得到相同的数据

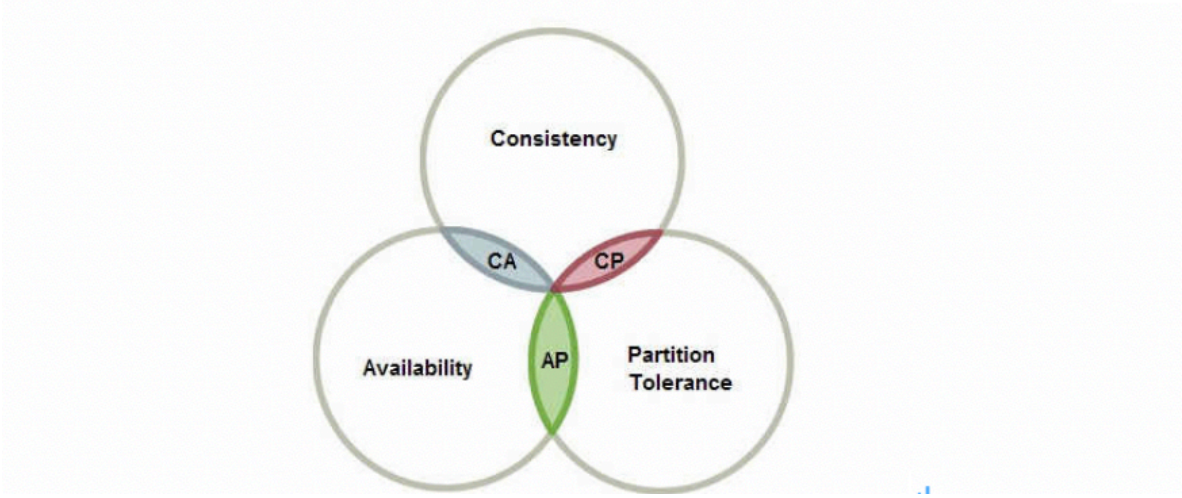
- 可用性：Availability

  可用性表示你总是能够访问集群（读/写），即使集群中的某个结点宕机了

- 分区容忍性：Partition toleranc

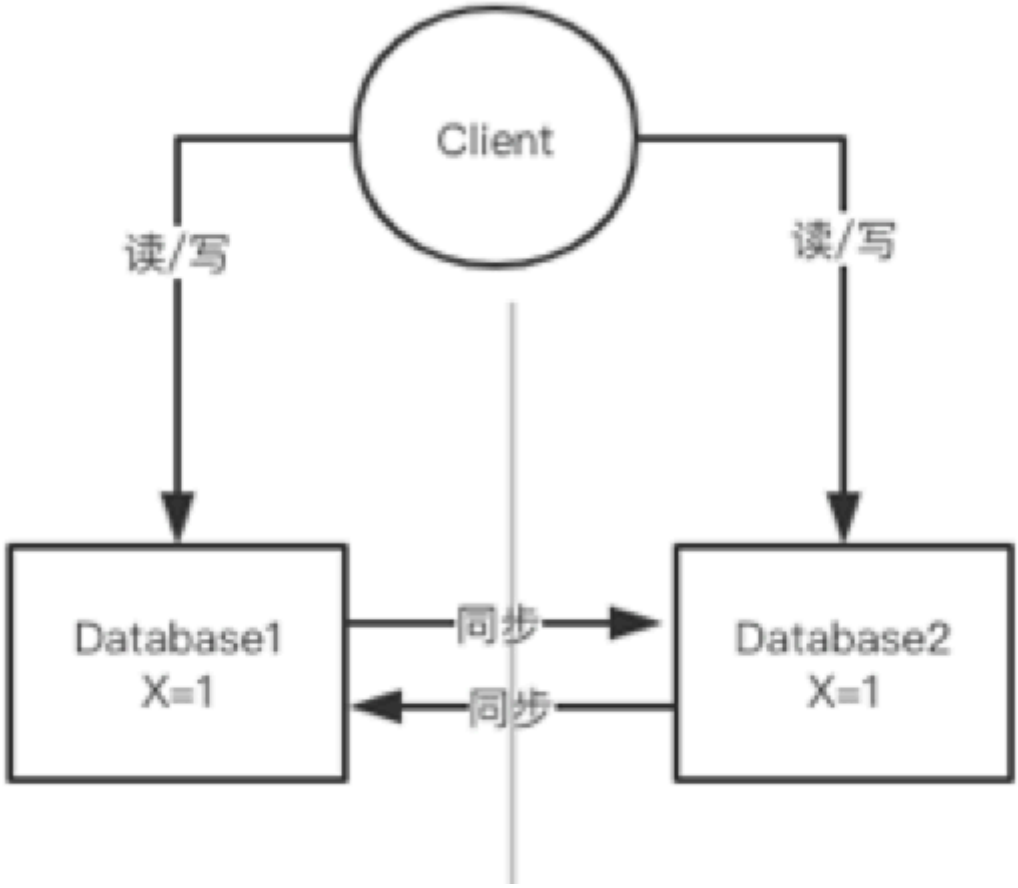
  容忍集群持续运行，即使他们中存在分区（两个分区中的结点都是好的，只是分区之间不能通信）

  分区就是指网络区域（就是说区域与区域之间不能）



结论：分布式环境下，CAP不能同时成立

3.1.2 模型解释



1. 如上图所示，假如DB1和DB2都能够正常被访问，只是他们之间不能够互相通信，也就是他们之间不能够同步数据，这个时候我们容忍集群继续运行，那么我们说服务具有分区容忍性
2. 那么现在在分区容忍性的前提之下：（DB1和DB2不能通信，服务继续运行）

   现在向DB1中发送一条X=2的更新请求，Datebase1把X值更新为2，但是需要去同步到Database2，由于DB1和DB2不能够通行，所以同步会失败

   A, 假如该条请求返回更新成功，那么会导致DB1和DB2数据不一致的情况出现，违背了一致性

   B, 假如该条请求返回更新失败，那么我们认为DB1不可用了，违背了可用性
3. 分布式系统在什么时候存在不满足分区容忍性的情况呢?

容忍集群持续运行，即使他们中存在分区 (两个分区中的结点都是好的，只是分区之间不能通信)
即P不存在，意思就是集群不能容忍分区的出现，也就是不能容忍两个节点之间不能通信的情况，而分布式环境下两个节点不能通信的情况是存在的，因为节点之间的通信都是通过网络传输，网络是不“靠谱”的。

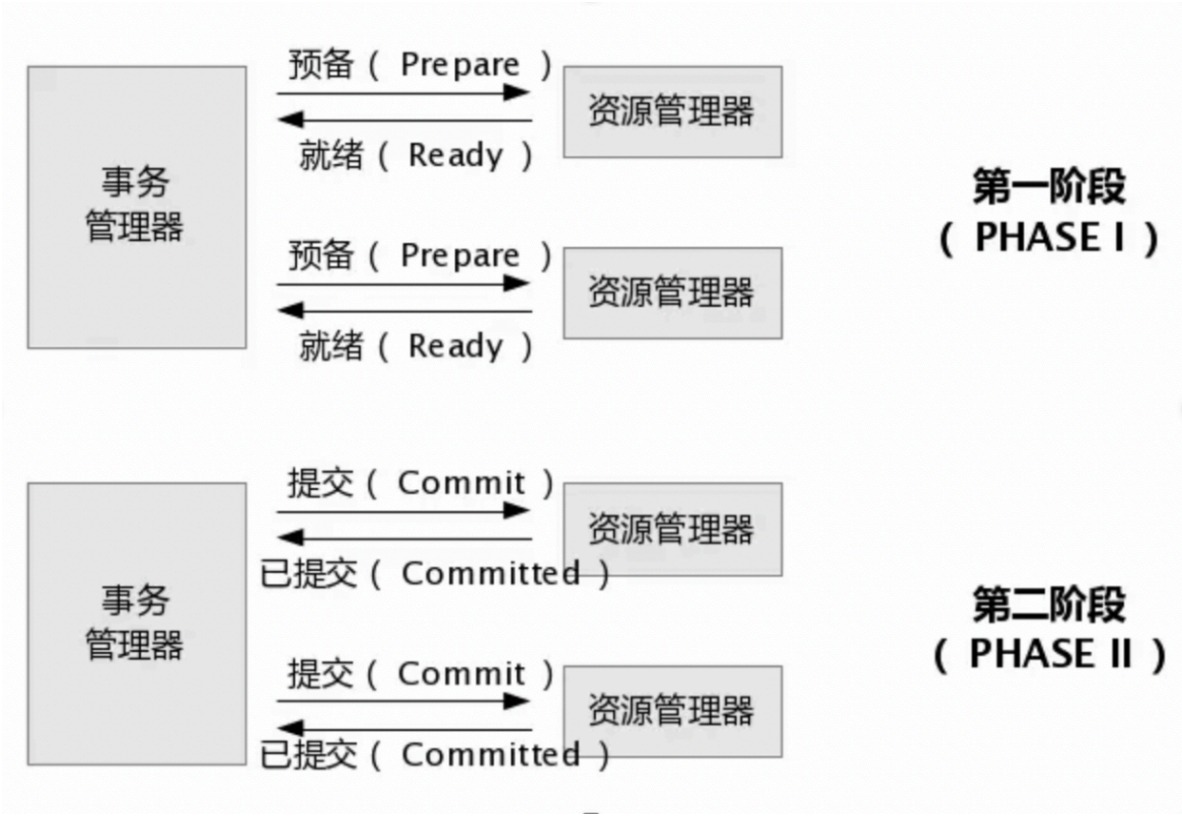
3.2 解决方案探索

3.2.1 刚性事务（强一致性）

定义：遵循ACID原则，强一致性。

代表：二阶段提交（2PC）

二阶段提交协议是协调所有分布式原子事务参与者，并决定提交或取消（回滚）的分布式算法。



1. 预备阶段

   在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。
   在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。

2. 提交阶段

   在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。参与者在接收到协调者发来的消息后将执行响应的操作。

存在的问题：

同步阻塞问题：在执行的过程中，所有参与的节点都是事务型阻塞的，当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态

不能解决数据不一致的问题

3.2.2 柔性事务（最终一致性）

定义

遵循BASE理论，最终一致性；与刚性事务不同，柔性事务允许一定时间内，不同节点的数据不一致，但要求最终一致。

Base理论

- Basically Avaiable，基本可用
- Soft state：软状态
- Eventually consistent：最终一致性

  既然无法做到强一致性，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性。

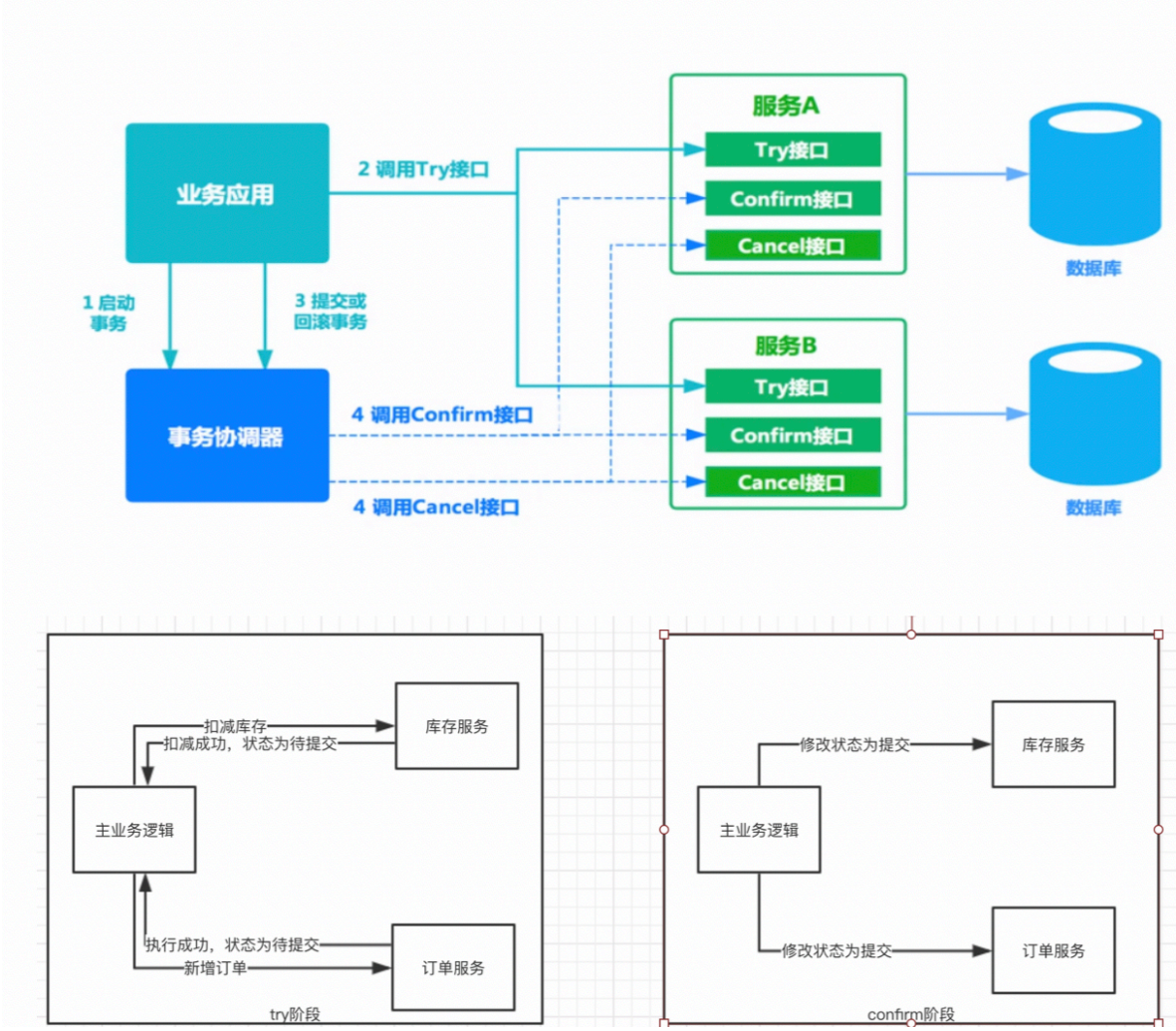
TCC事务

全称：Try-Confirm-Cancel（可以理解为sql中的Prepare、Commit、Rollback）

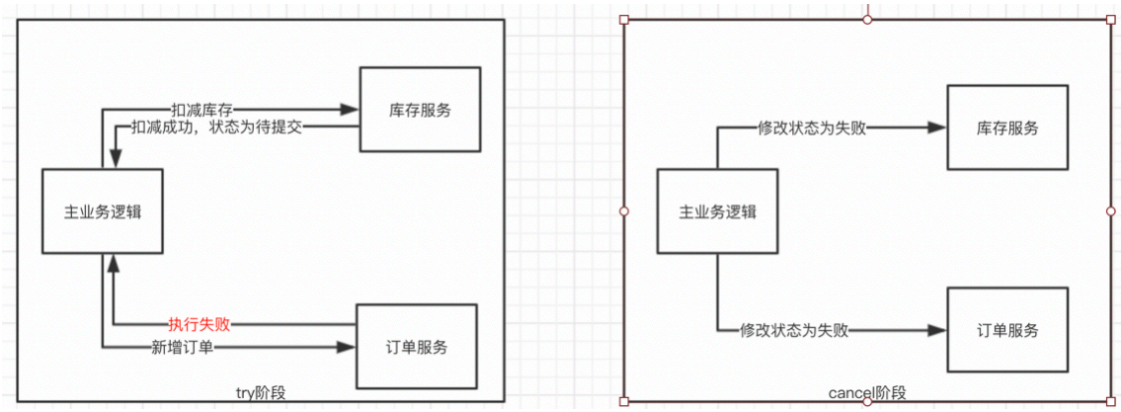
TCC是服务化的二阶段编程模型：

其 Try、Confirm、Cancel 3 个方法均由业务编码实现：Try 操作作为一阶段，负责资源的检查和预留。Confirm 操作作为二阶段提交操作，执行真正的业务。Cancel 是预留资源的取消。

事务流程



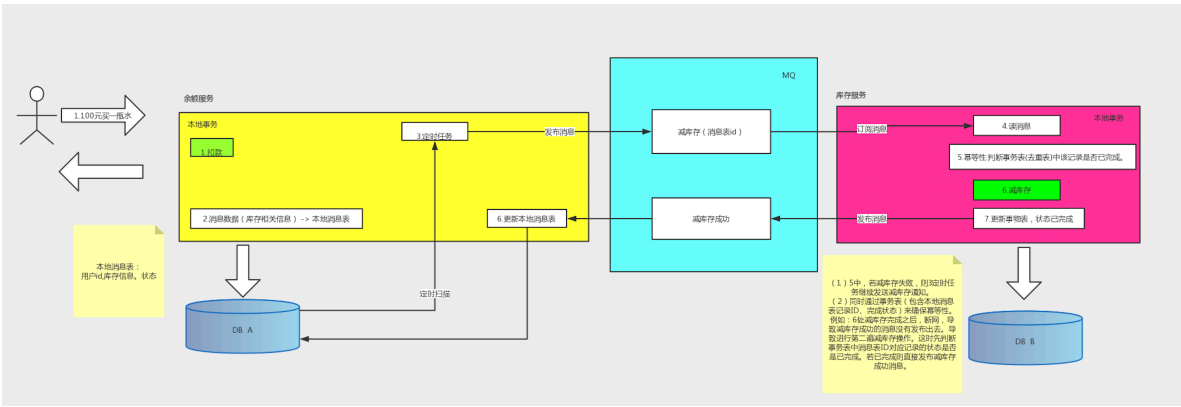
- Try阶段：完成所有的业务检查，预留资源
- Confirm阶段：更改状态操作
- Cancel阶段：当Try阶段存在服务执行失败时，则进入Cancel阶段



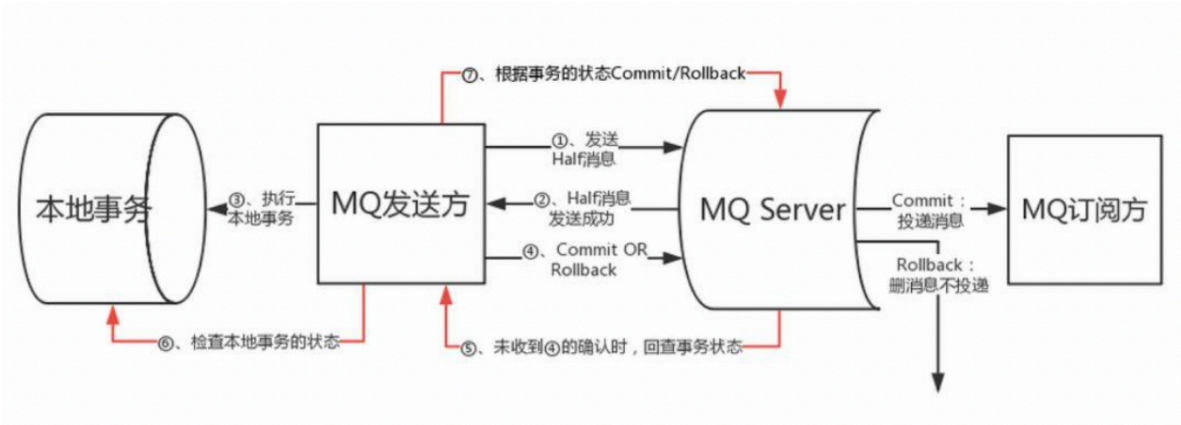
缺点：TCC的Try、Confirm、Cancel操作功能按照具体业务来实现，业务耦合度高，开发成本高

本地消息表

本地消息表这个方案最初是ebay提出的分布式事务完整方案



RocketMQ事务消息



1. 发送事务型消息，该消息暂时不可被消费
2. 消息队列返回发送消息成功状态
3. 本地事务收到消息发送成功状态，然后开始执行本地事务
    - 假如本地事务执行成功，MQ消息发送方则会告诉MQServer表示这个消息可以被消费了，MQ会投递这个消息到MQ订阅方
    - 假如本地事务执行失败，MQ消息发送方则会告诉MQServer需要丢弃之前发送的消息
4. 假如MQServer一直没有收到MQ发送方的消息确认通知，会回查本地事务，查看本地事务是否执行成功，假如本地事务执行成功，那么会发送消息，假如本地事务执行失败，那么会丢弃事务，假如本地事务还在初始态，那么会过一会再来询问

5. 消息在MQ订阅方的消息消费成功由MQ来保证

如何保证？（记下来，要考）

- MQ假如投递消息到MQ订阅方失败了，或者MQ订阅方消费消息失败了，那么MQ会把该消息丢入重试队列中，会重试发送该消息，默认16次，直到消息被消费成功为止
- 假如在16次之后该消息还没有被消费成功，那么MQ会再次把该消息丢入MQ死信队列中，对于死信队列的消息，我们需要手动去干预，让他消费成功（例如从后台管理系统手动（或者是定时任务）把死信队列中的消息拿出来，然后手动去执行操作，执行完成之后把消息从死信队列中删除掉）

3.4 事物框架介绍

- **阿里GTS**

成熟的方案，一站式解决，需要付费，参考链接：https://helpcdn.aliyun.com/product/48444.html

- **基于GTS的免费社区版本，SEATA**

参考链接：https://github.com/seata/seata

```
SEATA有不同的模式，分为AT模式，TCC模式，XA模式，SAGA模式：
a. AT模式：XA/2PC协议实现的变种实现，它支持事物的回滚，但是回滚本身是由SEATA实现，并非数据库的自动回滚
b. TCC模式：基于TCC模式，可以实现TCC事物
c. XA模式：基于XA模式，可以实现XA/2PC协议的2阶段提交事物
```

- 其他的事物框架，比如Atomikos，LCN，以及TCC-Transaction，ByteTcc等等

4 消息型事务代码实现

消息生产者代码

```
    public static void main(String[] args) throws MQClientException {
        // 1.创建一个用于发送事物消息的一个Producer对象
        TransactionMQProducer producer = new
TransactionMQProducer("32th_transaction_producer");

        //2. 设置nameserver地址
        producer.setNamesrvAddr("127.0.0.1:9876");

        //3. 设置事物监听器
        producer.setTransactionListener(new TransactionListener() {

            /*
                  在该方法中执行上游服务的本地事物， 对应到我们的秒杀服务，就是扣减库存
             */
            @Override
            public LocalTransactionState executeLocalTransaction(Message msg,
Object arg) {
                // 每一个事物消息，都有和本次事物对应的一个 tansactionId
                System.out.println("executeLocalTransaction 执行了： " +
msg.getTransactionId());
                return LocalTransactionState.UNKNOW;
            }
```

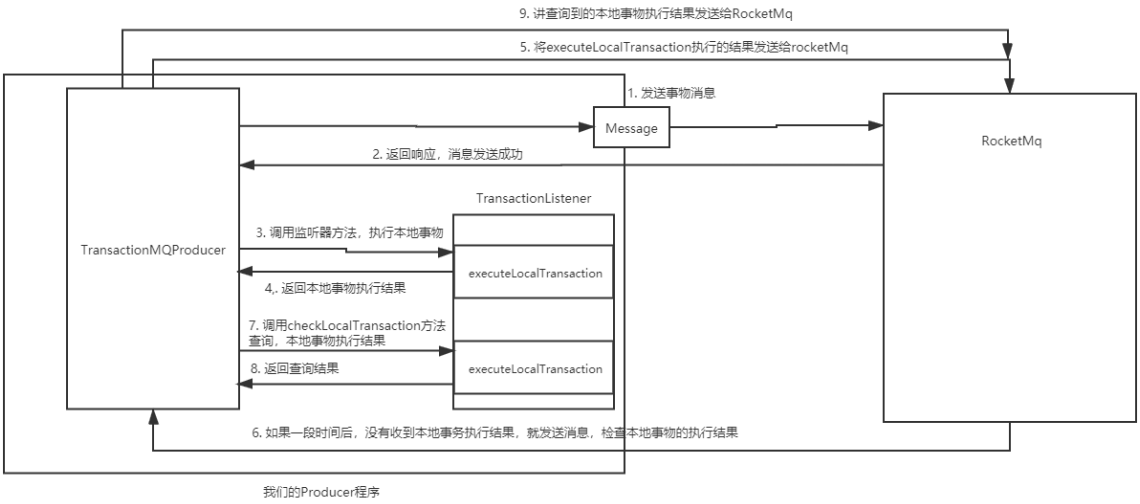
```
            @Override
            public LocalTransactionState checkLocalTransaction(MessageExt msg) {
                System.out.println("checkLocalTransaction 执行了: " +
msg.getTransactionId());
                return LocalTransactionState.UNKNOW;
            }
        });

        //4. 启动producer
        producer.start();

        //5. 准备待发送的消息
        Message message = new Message();
        message.setTopic("test_transaction");
        String data = "hello, transaction message";
        message.setBody(data.getBytes(Charset.forName("utf-8")));

        //6. 发送事物消息
        TransactionSendResult sendResult =
producer.sendMessageInTransaction(message, null);
        System.out.println(sendResult);
    }
```

TransactionListener的中两个方法的执行时机



消息消费者代码

```
    public static void main(String[] args) throws MQClientException {

        DefaultMQPushConsumer consumer
                = new DefaultMQPushConsumer("32th_transaction_consumer");

        consumer.setNamesrvAddr("127.0.0.1:9876");

        consumer.subscribe("test_transaction", "*");

        consumer.setMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs, ConsumeConcurrentlyContext context) {
```

```
                MessageExt message = msgs.get(0);
                byte[] body = message.getBody();
                String data = new String(body, 0, body.length,
Charset.forName("utf-8"));
                System.out.println("data: " + data + "---" +
message.getTransactionId());
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });

        consumer.start();
    }
```

5.实现秒杀下单的分布式事务

- 在秒杀服务中，准备好封装了订单服务下单所需参数的Request对象(因为秒杀下单的商品价格是秒杀价格，所以在此之前，还需先查询出秒杀商品的秒杀价格)

- 在秒杀服务中，准备好一个用于发送事物消息的Producer对象，在启动Producer之前，设置好TransactionListener，在executeLocalTransaction方法中实现秒杀商品库存的扣减(即执行本地事物)，在checkLocalTransaction中实现，对于本地事物执行结果的查询逻辑

- 在订单服务中，将订单服务下单所需的Request对象，封装到待发送的事物消息的消息体中，并指定事物消息的主题，将事物消息发送到RocketMQ

- 在订单服务中，实现一个用于消费事物消息的消费者，在接受到订单服务发送的，用于秒杀下单的事物消息之后，取出相关的下单数据，即Request对象，调用订单服务相关接口完成秒杀下单的功能

6 秒杀下单中的问题二

我们的项目汇中，可能存在着多个秒杀服务的实例，即多个用户可能访问不同的秒杀服务，同时秒杀下单，那么这里就会有一个潜在的问题了。

```
# 秒杀下单的核心sql语句如下
item_stock = item_stock - 1
而这条sql语句的执行可以分成3步：
1）读取item_stock旧值
2）item_stock旧值 - 1
3) 给item_stock赋予新值
```

所以，可能会出现秒杀商品的超卖问题，那么如何解决这个问题呢？解决超卖问题需要加锁，但是使用synchronized，或者JDK的Lock锁，显然是不行的，因此我们只能使用分布式锁，分布式锁的实现有多种方式，比较常见的有基于MySQL的实现，基于Zookeeper的实现，以及基于Redis的实现。

我们下面就来看看基于Redis实现的分布式锁，如何使用

```
// 定义一把锁在Redis中对应的key
String lockKey = "promo_session_item:" + productId + "-" + psId;
// 通过定义好的key，获取这把分布式锁
RLock lock = redissonClient.getLock(lockKey);
// 加锁
lock.lock();
try {
  执行需要同步的代码
}  finally {
   // 释放锁
   lock.unlock();
 }
```