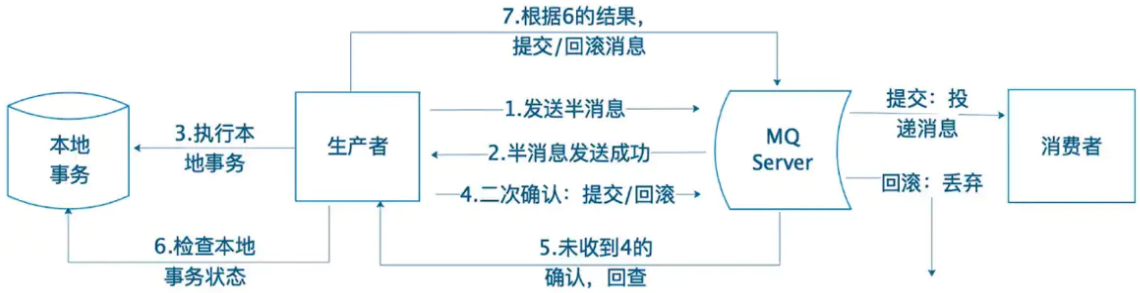
如何用RocketMQ实现分布式事务

2020.08.15 19:22:56字数 1,327阅读 183

本篇文章我们会以秒杀场景为例演示如何利用RocketMq实现分布式事务。
开始之前我们先来了解rocketmq的事务消息设计和流程



流程

Apache RocketMQ在4.3.0版中已经支持分布式事务消息，这里RocketMQ采用了2PC的思想来实现了提交事务消息，同时增加一个补偿逻辑来处理二阶段超时或者失败的消息

一、RocketMQ事务消息流程概要

- **事务消息发送及提交：**

> 1. 发送消息（half消息）。此阶段的消息对任何消费端不可见，不会被发现和消费。消息被存储在Broker中。
> 2. 服务端响应消息写入结果。
> 3. half消息发送成功执行本地事务方法
> 4. 根据本地事务状态执行Commit或者Rollback，如果提交状态为Commit这Broker将half消息标记为可消费状态，
>
> 此时消费端就可以消费该条消息；如果是Rollback，则Broker会将该消息删除

如果第4步的二次确认消息因为事故发送失败，比如服务挂了，停电等，broker不知道是commit还是rollback，那么rocketmq需要定时回查生产者，确认消息状态。

- **补偿流程：**

> 5. 对没有Commit/Rollback的事务消息（pending状态的消息），从服务端发起一次“回查”
> 6. Producer收到回查消息，检查回查消息对应的本地事务的状态
> 7. 根据本地事务状态，重新Commit或者Rollback

其中，补偿阶段用于解决消息Commit或者Rollback发生超时或者失败的情况。

二、开始编码实战

在秒杀业务当中系统的性能瓶颈来自于数据库，秒杀时所有的用户会同时抢一件商品，从技术讲其实就是所有请求竞争同一条行锁，在高并发时，都在竞争同一行锁，所有请求串行化，会大大降低系统的吞吐量。

其中一种方案就是缓存库存，异步扣减，操作内存的速度要大大超过操作磁盘。

1. 活动发布同步库存进缓存
2. 下单交易扣减缓存中库存
3. 发送mq异步扣减数据库库存

我们先来看看发送普通消息会有什么问题
1.引入依赖

```
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.15.RELEASE</version>
  </parent>
 <dependency>
      <groupId>org.apache.rocketmq</groupId>
      <artifactId>rocketmq-spring-boot-starter</artifactId>
      <version>2.1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
```

2.配置地址
rocketmq.name-server=192.168.1.8:9876
rocketmq.producer.group=producer_group

spring.redis.host=127.0.0.1
spring.redis.port=6379
spring.redis.database=5

3.下单

```
    @Override
    @Transactional
    public OrderModel createOrder(Integer userId, Integer itemId, Integer
promoId, Integer amount) throws BusinessException {
        //1.校验下单状态,下单的商品是否存在，用户是否合法，购买数量是否正确
        ItemModel itemModel = itemService.getItemById(itemId);
        if(itemModel == null){
            throw new
BusinessException(EmBusinessError.PARAMETER_VALIDATION_ERROR,"商品信息不存在");
        }

        UserModel userModel = userService.getUserById(userId);
        if(userModel == null){
```

```
            throw new
BusinessException(EmBusinessError.PARAMETER_VALIDATION_ERROR,"用户信息不存在");
        }
        if(amount <= 0 || amount > 99){
            throw new
BusinessException(EmBusinessError.PARAMETER_VALIDATION_ERROR,"数量信息不正确");
        }

        //校验活动信息
        if(promoId != null){
            //（1）校验对应活动是否存在这个适用商品
            if(promoId.intValue() != itemModel.getPromoModel().getId()){
                throw new
BusinessException(EmBusinessError.PARAMETER_VALIDATION_ERROR,"活动信息不正确");
                //（2）校验活动是否正在进行中
            }else if(itemModel.getPromoModel().getStatus().intValue() != 2) {
                throw new
BusinessException(EmBusinessError.PARAMETER_VALIDATION_ERROR,"活动信息还未开始");
            }
        }

        //2.落单减库存，从缓存中扣减
        boolean result = itemService.cacheDecreaseStock(itemId, amount);
        if(!result){
            throw new BusinessException(EmBusinessError.STOCK_NOT_ENOUGH);
        }

        //3.订单入库
        OrderModel orderModel = new OrderModel();
        orderModel.setUserId(userId);
        orderModel.setItemId(itemId);
        orderModel.setAmount(amount);
        if(promoId != null){

 orderModel.setItemPrice(itemModel.getPromoModel().getPromoItemPrice());
        }else{
            orderModel.setItemPrice(itemModel.getPrice());
        }
        orderModel.setPromoId(promoId);
        orderModel.setOrderPrice(orderModel.getItemPrice().multiply(new
BigDecimal(amount)));

        //生成交易流水号,订单号
        orderModel.setId(generateOrderNo());
        OrderDO orderDO = convertFromOrderModel(orderModel);
        orderDOMapper.insertSelective(orderDO);

        //加上商品的销量
        itemService.increaseSales(itemId,amount);
        //4.返回前端
        return orderModel;
    }
```

4.减库存的逻辑

```
 public boolean cacheDecreaseStock(Integer itemId, Integer amount) throws
BusinessException {
        // 扣减缓存库存
        long result =
redisTemplate.opsForValue().increment("promo_item_stock_"+itemId,amount.intValue
() * -1);
        if(result > 0) {
            // 缓存的库存扣减成功,发送异步消息扣减库存
            try {
                rocketMQTemplate.convertAndSend("seckill",

 DecreaseStockMessage.builder().itemId(itemId).amount(amount).build());
            } catch(MessagingException e) {
                log.error("异步扣减库存消息发送异常", e);
                // 发送消息异常库存需要回填
                increaseStock(itemId, amount);
                return false;
            }
            //更新库存成功
            return true;
        } else if(result == 0) {
            //打上库存已售罄的标识
            redisTemplate.opsForValue().set("promo_item_stock_invalid_" +
itemId, "true");

            //更新库存成功
            return true;
        } else {
            // 更新库存失败,缓存库存回填
            increaseStock(itemId, amount);
        }
        return false;
    }
```

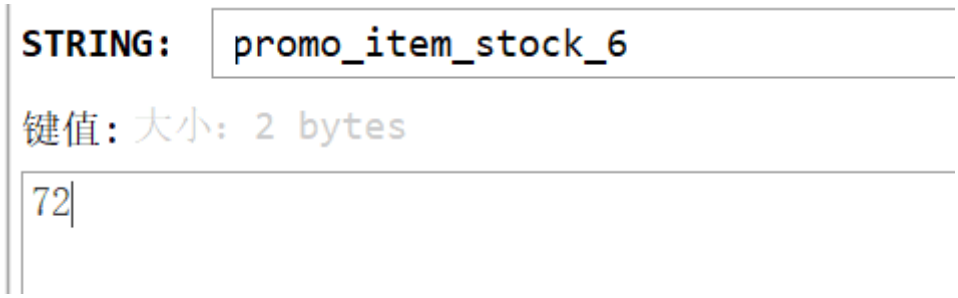
4.消费端代码

```
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class DecreaseStockMessage {

    private Integer itemId;
    private Integer amount;
}


@Component
@RocketMQMessageListener(consumerGroup = "consumer_group", topic = "seckill")
public class DecreasePromotStockListener implements
RocketMQListener<DecreaseStockMessage> {

    @Autowired
    private ItemStockDOMapper stockDOMapper;
```

```
    @Override
    public void onMessage(DecreaseStockMessage decreaseStockMessage) {
        Integer amount = decreaseStockMessage.getAmount();
        Integer itemId = decreaseStockMessage.getItemId();
        stockDOMapper.decreaseStock(itemId, amount);
    }
}
```

执行前数据库库存和缓存中的库存

id	stock	item_id
6	71	6

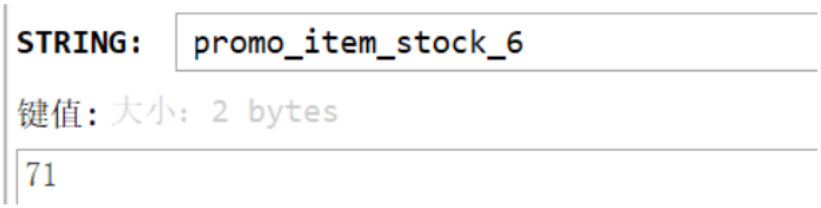


数据库库存

id	stock	item_id
6	70	6

执行后数据库库存和缓存中的库存

id	stock	item_id
6	70	6



结果是成功的，貌似好像万无一失，可果真是如此吗，我们来分析一下。

下单的逻辑是在异步扣减库存后去生成订单，那么如果下单失败了可是库存已经异步扣减了，那么就造成了数据的不一致，比如用户取消支付等
没有办法去回补库存

这就会造成少卖的现象，商家会发现库存莫名奇妙的减少了可是找不到对应的订单
本质就是一个分布式事务的问题，因为@Transactional并不能保证异步扣减库存和生成订单同时成功或失败。

那么这样的话，一个改造就是将发送消息延后到创建订单后，整个下单方法最后返回之前，比如这样

```
// ........省略以上生成订单等逻辑,代码已在上面贴出

        // 异步减库存
         boolean b = itemService.asyncDecreaseStock(itemId, amount);
         if(!b) {
             // 回补redis的库存
             itemService.increaseSales(itemId, amount);
         }
         //4.返回前端
         return orderModel;
```

但是这样其实也是会有问题的，消息发送成功，整个下单顺利执行完成，但是最后spring的声明式事务最后提交前突然发生一些事故，比如网络超时，服务宕机等等导致事务没有提交，但是消息已经发送成功，库存还是会丢掉。

那么这种时候的解决办法就是需要有一种方式可以让事务真正提交成功后发送消息，而且消息发送失败的情况下依然有补偿措施。接下来我们就看看通过rocketmq怎么解决这一问题。

三、引入RocketMq事务消息

那么当有下单请求进来时，直接发送第一阶段消息，如下

```
@Override
    public boolean asyncDecreaseStockTransaction(Integer userId, Integer
promoId, Integer itemId, Integer amount) {
        try {
            Map<String, Integer> argsMap = new HashMap<>();
            argsMap.put("userId", userId);
            argsMap.put("promoId", promoId);
            argsMap.put("itemId", itemId);
            argsMap.put("amount", amount);

            String s = JSONUtil.toJsonPrettyStr(argsMap);

            rocketMQTemplate.sendMessageInTransaction(
                    "tx_producer_group",
                    "seckill",
                    MessageBuilder.withPayload(

 DecreaseStockMessage.builder().itemId(itemId).amount(amount).build())
                            .build(),
                    s);
        } catch(MessagingException e) {
            return false;
        }
        return true;
    }
```

第一阶段消息发送成功会执行定义的本地事务，在这里去调用上面贴出的下单方法，如下：

```
@Slf4j
@Component
@RocketMQTransactionListener(txProducerGroup = "tx_producer_group")
public class DecreasePromoStockTransactionListener implements
RocketMQLocalTransactionListener {

    @Autowired
    private OrderService orderService;


    /**
     * 发送第一阶段消息成功后执行本地事务
     * 生成订单
     * @param message
     * @param o
     * @return
     */
    @Override
    public RocketMQLocalTransactionState executeLocalTransaction(Message
message, Object o) {
        log.info("开始执行本地事务message：{}, args:{}", message, o);
        try {
            Object payload = message.getPayload();
            String jsonStr = (String) o;
            HashMap hashMap = JSONUtil.toBean(jsonStr, HashMap.class);
            Integer userId = (Integer) hashMap.get("userId");
            Integer itemId = (Integer) hashMap.get("itemId");
            Integer promoId = (Integer) hashMap.get("promoId");
            Integer amount = (Integer) hashMap.get("amount");

            OrderModel order = orderService.createOrder(userId, itemId, promoId,
amount);
        } catch(Exception e) {
            log.error("本地事务异常", e);
            return RocketMQLocalTransactionState.ROLLBACK;
        }
        return RocketMQLocalTransactionState.COMMIT;
    }

    @Override
    public RocketMQLocalTransactionState checkLocalTransaction(Message message)
{
        log.info("回查开始{}", message);
        return RocketMQLocalTransactionState.COMMIT;
    }
}
```

扣库存的方法修改为：

```
public boolean cacheDecreaseStock(Integer itemId, Integer amount) throws
BusinessException {
        // 扣减缓存库存
        long result =
redisTemplate.opsForValue().increment("promo_item_stock_"+itemId,amount.intValue
() * -1);
        if(result > 0) {
            // 缓存的库存扣减成功,发送异步消息扣减库存
            //asyncDecreaseStock(itemId, amount);
            //更新库存成功
            return true;
        } else if(result == 0) {
            //打上库存已售罄的标识
            redisTemplate.opsForValue().set("promo_item_stock_invalid_" +
itemId, "true");
            //更新库存成功
            return true;
        } else {
            // 更新库存失败
            increaseStock(itemId, amount);
        }

        return false;
    }
```

回查方法还没有实现，为了在回查方法中追踪该笔交易，需要设计一张订单日志表

```
CREATE TABLE `stock_log` (
  `stock_log_id` varchar(64) NOT NULL,
  `item_id` int NOT NULL DEFAULT '0',
  `amount` int NOT NULL DEFAULT '0',
  `status` int NOT NULL DEFAULT '0' COMMENT '//1表示初始状态，2表示下单扣减库存成功，3
表示下单回滚',
  `create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`stock_log_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

1、只需要在发送第一阶段消息之前生成一条日志，插入数据库
2、生成订单后，更新日志的状态
3、将logId作为message的一个字段发送出去，在回查的方法中就可以获取到
4、查询log日志状态返回对应的状态

```
@Override
public LocalTransactionState checkLocalTransaction(MessageExt msg) {
    //根据是否扣减库存成功，来判断要返回COMMIT,ROLLBACK还是继续UNKNOWN
    String jsonString  = new String(msg.getBody());
    Map<String,Object>map = JSON.parseObject(jsonString, Map.class);
    Integer itemId = (Integer) map.get("itemId");
    Integer amount = (Integer) map.get("amount");
    // 从message消息中拿到日志id
    String stockLogId = (String) map.get("stockLogId");
    // 查询日志
```

```
    StockLogDO stockLogDO = stockLogDOMapper.selectByPrimaryKey(stockLogId);
    if(stockLogDO == null){
        return LocalTransactionState.UNKNOW;
    }
    if(stockLogDO.getStatus().intValue() == 2){
        return LocalTransactionState.COMMIT_MESSAGE;
    }else if(stockLogDO.getStatus().intValue() == 1){
        return LocalTransactionState.UNKNOW;
    }
    return LocalTransactionState.ROLLBACK_MESSAGE;
}
```