创新梦工厂商城-秒杀模块设计

👉 本文档记录了创新梦工厂商城-秒杀模块的核心设计，主要包含以下几个方面：

1. 幂等性处理
2. 分布式事务
3. 定时关单（延时队列）
4. 流量削峰

一、请求幂等性

为什么需要幂等?

1. 用户多次点击按钮
2. 页面回退再提交
3. 微服务互相调用，由于网络问题，导致请求失败，feign 触发重试机制
4. 其他业务情况

验证令牌的原子性

1. 第一次请求带上指定令牌（特殊数字串或者字符串）
2. 第二次请求的时候，在 redis + lua 中验证令牌是否重复（相等），若相等，则删除 Redis 中的对应数据
3. 若删除成功，则说明是第一次请求；否则是重复请求

```
1  //1、验证令牌是否合法【令牌的对比和删除必须保证原子性】
2  String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call
3  String orderToken = vo.getOrderToken();
4
5  //通过lure脚本原子验证令牌和删除令牌
6  Long result = redisTemplate.execute(new DefaultRedisScript<Long>(script, Long.cl
7          Arrays.asList(USER_ORDER_TOKEN_PREFIX + memberResponseVo.getId()),
8          orderToken);
9
10 if (result == 0L) {
11     //令牌验证失败，说明是重复请求
12     responseVo.setCode(1);
```

```
13      return responseVo;
14  } else {
15      //令牌验证成功 删除成功，说明是第一次请求
16
17      ......
18  }
```

二、RabbitMQ 实现分布式柔性事务

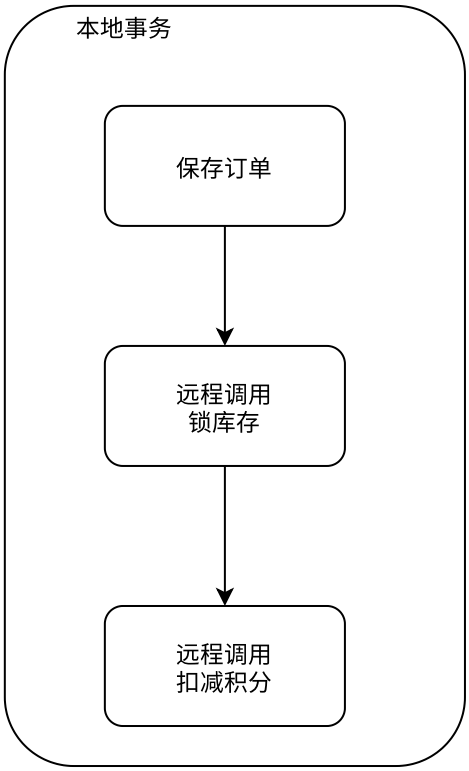
本地事务的缺点

现在的基本流程：

以下三个节点是属于同一个本地事务内的；

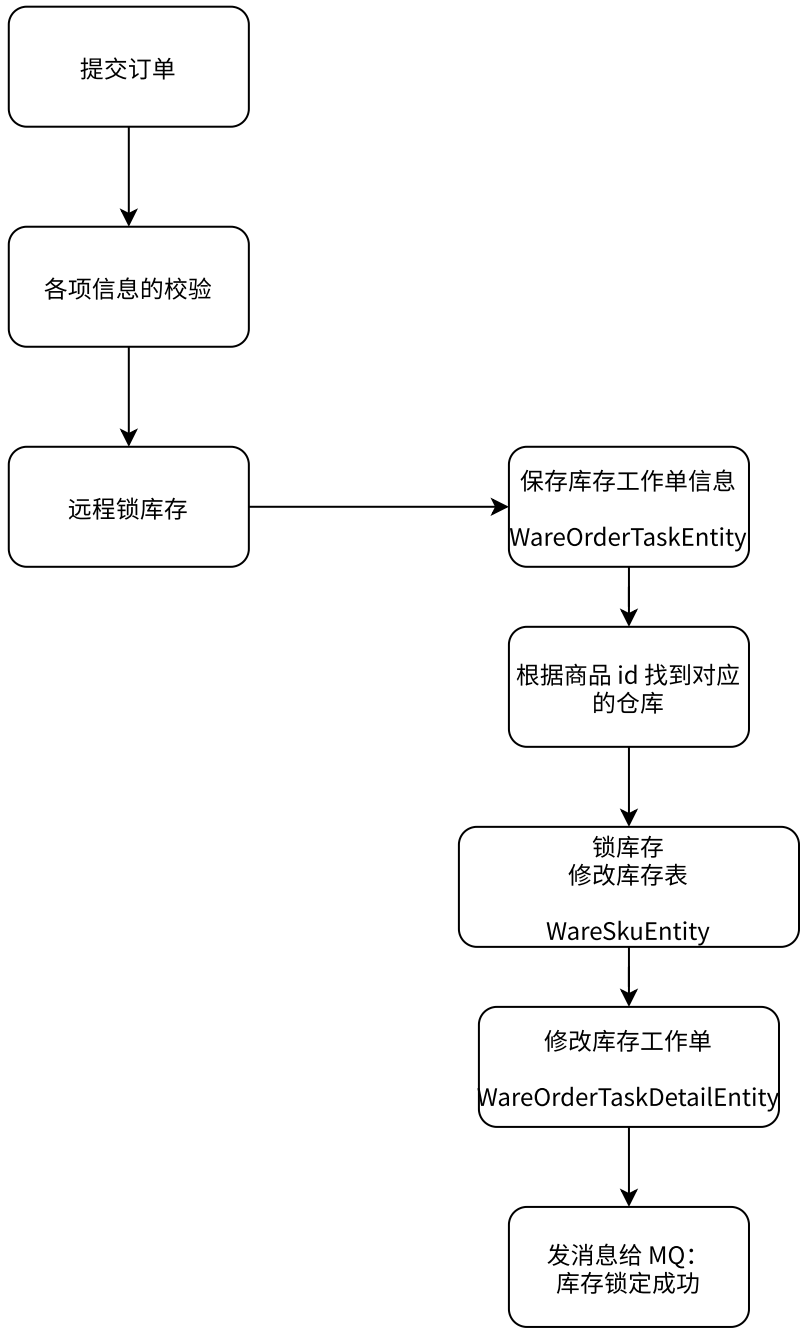
如果在保存订单节点出现问题，本地事务会回滚；

如果是在下面两个远程调用的结点中出现问题，本地事务也会回滚；但是本地事务回滚的只是保存订单节点的操作，而对应的远程调用结点的操作将无法回滚；也就是说本地事务无法让所有环境回滚到操作前。



MQ 实现分布式事务-最终一致性

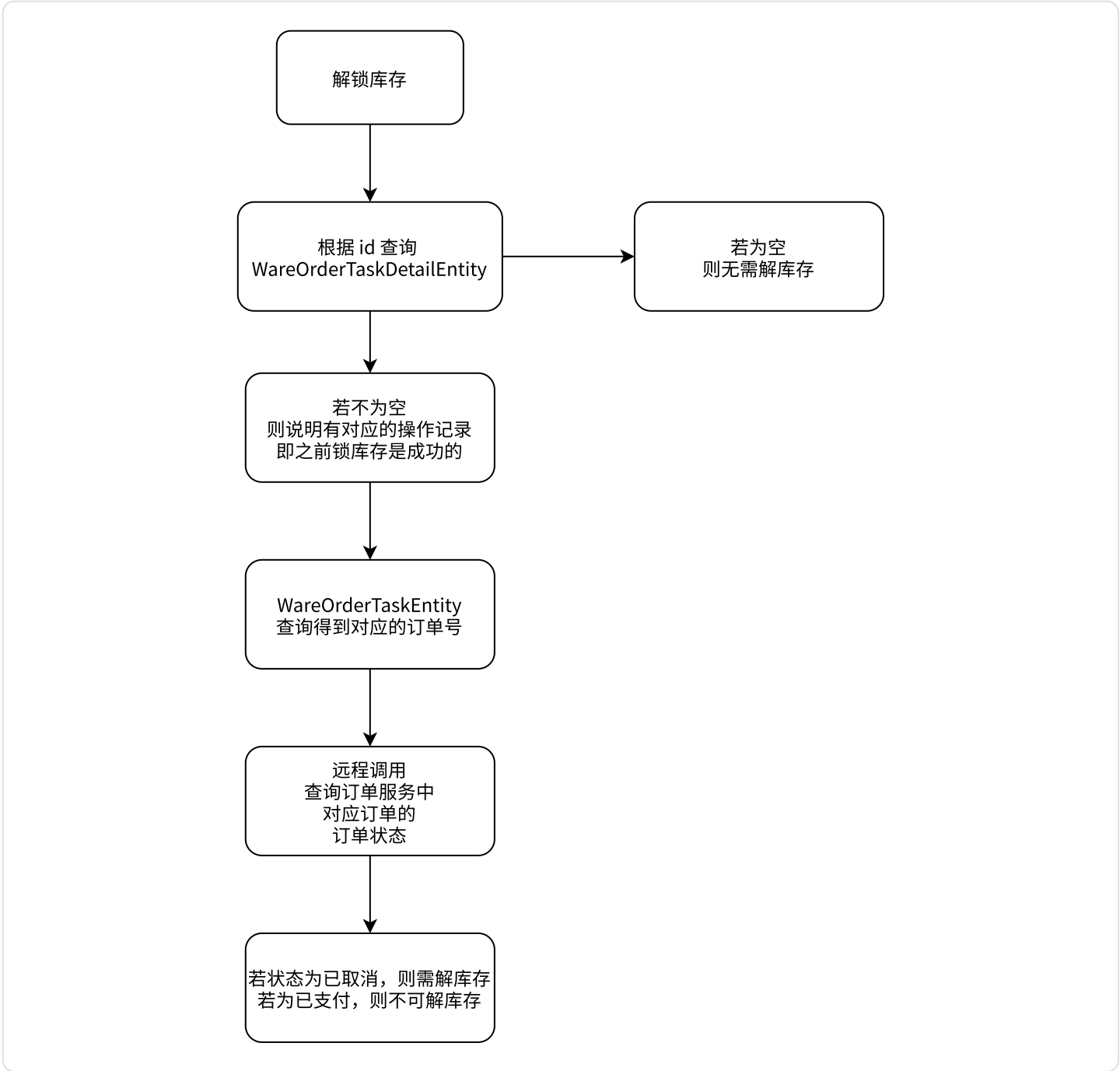
锁库存的流程：



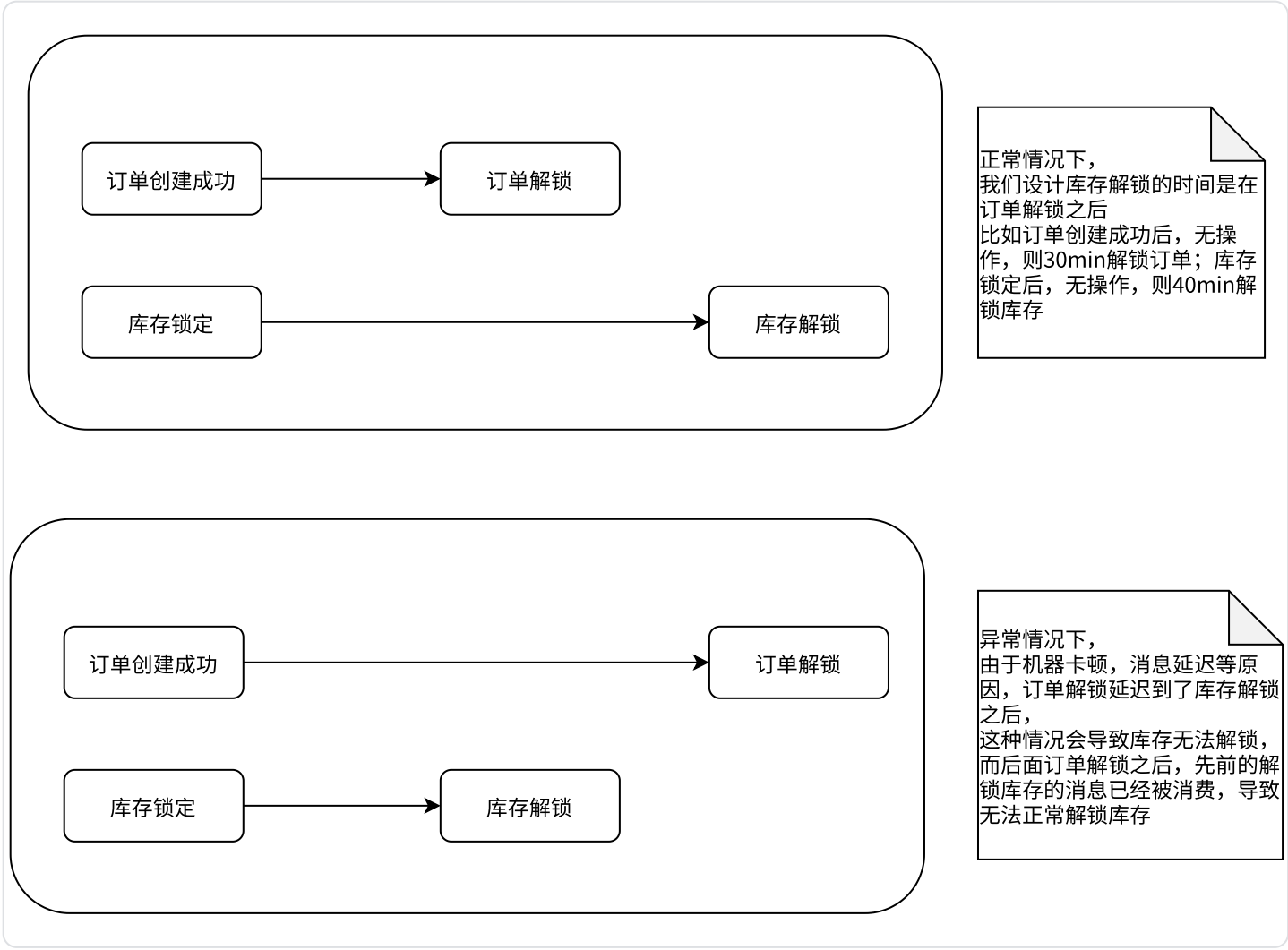
解库存的流程：

库存解锁的情况：

1. 下订单成功，库存锁定成功，接下来的业务调用失败，导致订单回滚。之前锁定的库存就要自动解锁



如果下订单成功了，则说明锁库存也成功了；所以在解库存之前需要判断订单已经关了：



假设订单创建后30min，订单解锁，那库存解锁可能得设计是订单创建后的40min（因为库存解锁的时候会去判断当前订单的状态，如果该状态还是未支付的话，则库存不解锁）

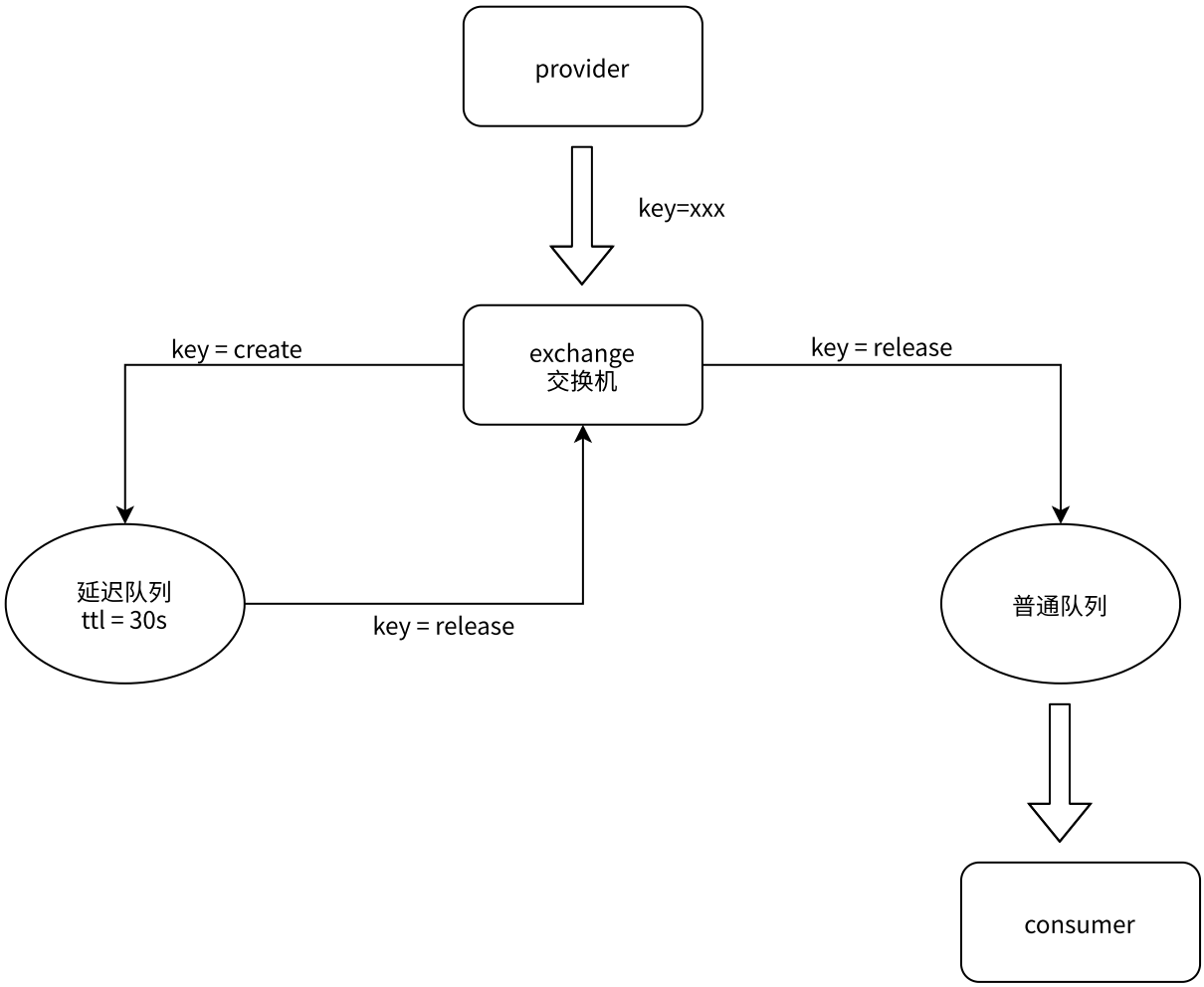
就算设计了时间差，但是因为卡顿等各种原因，有可能会导致订单解锁在订单创建后40min内没有解锁成功的；这时候库存解锁时会发现订单状态为未支付，无需解库存，但是当前的消息却会被消费；导致后面订单被解锁后，而库存无法解锁。

为了解决上述问题，在订单解锁成功之后，需要再发一个解锁库存的 MQ 消息，待库存服务消费（解库存）

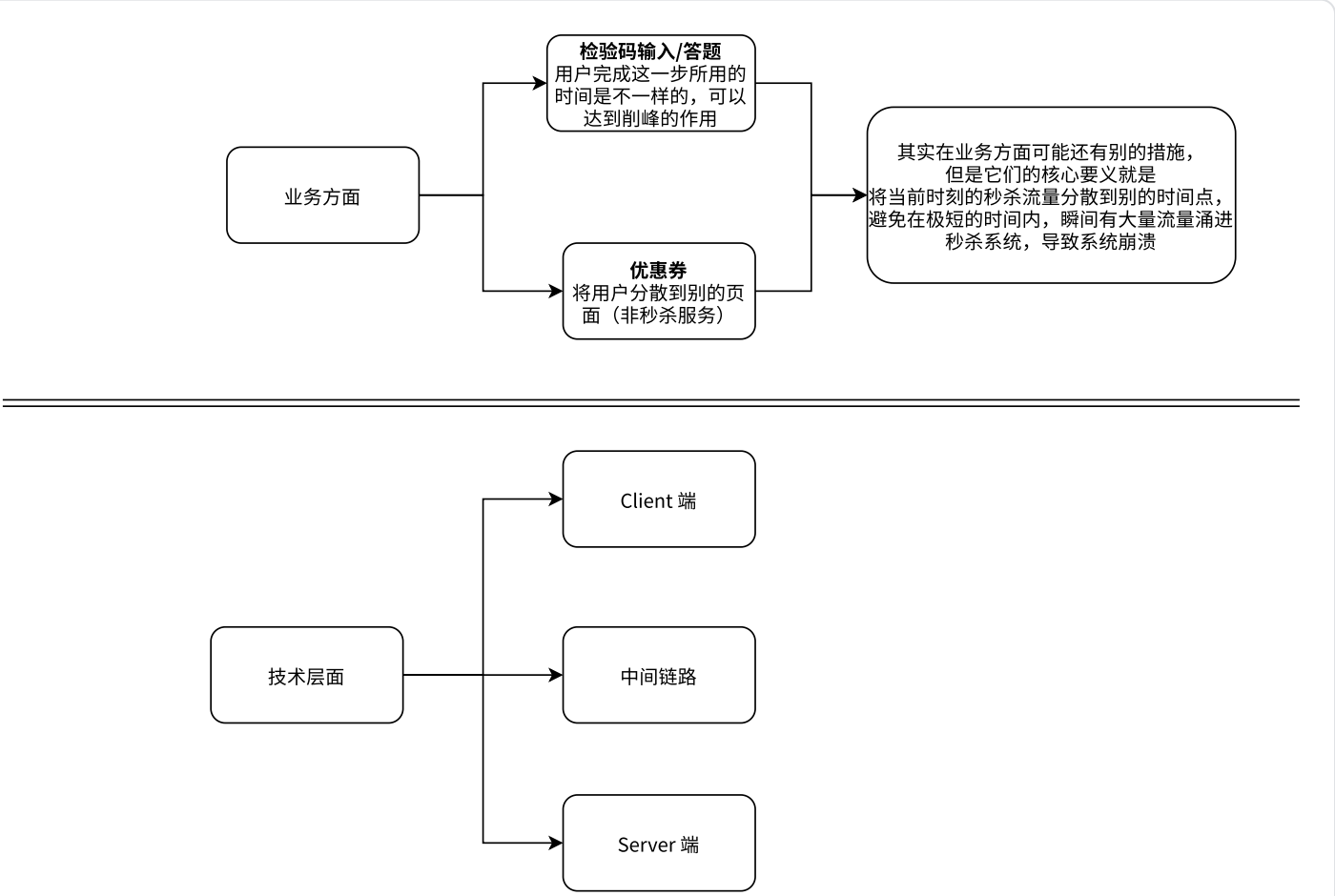
三、RabbitMQ 实现定时关单-延时队列

解决了定时任务的时效性问题

解决事务最终一致性



四、流量错峰 vs 流量削峰

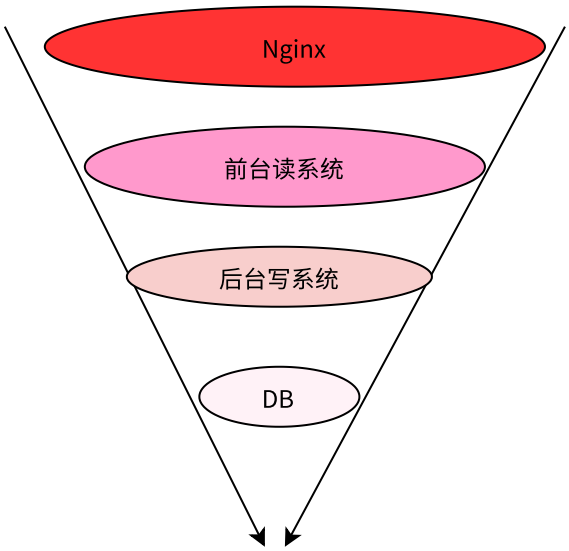


业务层面

1. 在业务层面可以在点击秒杀按钮之前，先进行验证码输入、答题等方式进行分散用户流量（不同用户的输入速度不一样，可以避免所有用户在同一时刻发出秒杀请求）
2. 优惠券（或者其他吸引用户的页面），在引流的同时也可以将用户分散到别的页面。

基于漏斗模型对数据进行分层过滤

分层过滤的核心思想是：在不同的层次尽可能地过滤掉无效请求，让“漏斗”最末端的才是有效请求。



要达到分层过滤的效果，我们就需要实现分层校验的逻辑：

1. 利用 Nignx 进行数据动静分离改造，这一层可以拦截大部分的静态数据请求；
2. 前台读系统可以走 Cache，并过滤掉一些无用请求；
3. 后台写系统，主要对写的数据（如“库存”）做一致性检查，需要对数据做二次检验，对系统做好保护和限流，这样数据量和请求就进一步减少；
4. 在数据库层需要保证数据的最终准确性（如“库存”不能减为负数）

防止非正常请求

在秒杀的 URL 添上随机码，防止直接调 API 秒杀。（随机码是只有在秒杀开始时，才会下发给用户，这样就可以避免提前脚本请求）

```
/**
 * 商品信息预热对象
 **/
@Data
public class SeckillSkuRedisTo {

    ......

    /**
     * 商品id
     */
    private Long skuId;
    /**
     * 秒杀价格
     */
    private BigDecimal seckillPrice;
    /**
```

```
18       * 秒杀总量
19       */
20      private Integer seckillCount;
21      /**
22       * 每人限购数量
23       */
24      private Integer seckillLimit;
25      /**
26       * 排序
27       */
28      private Integer seckillSort;
29
30      //sku的详细信息
31      private SkuInfoVo skuInfo;
32
33      //当前商品秒杀的开始时间
34      private Long startTime;
35
36      //当前商品秒杀的结束时间
37      private Long endTime;
38
39      // 当前商品秒杀的随机码
40      // 只有在秒杀的时候才会返回 randomCode，其他情况再返回之前需要去除 randomCode
41      private String randomCode;
42  }
```