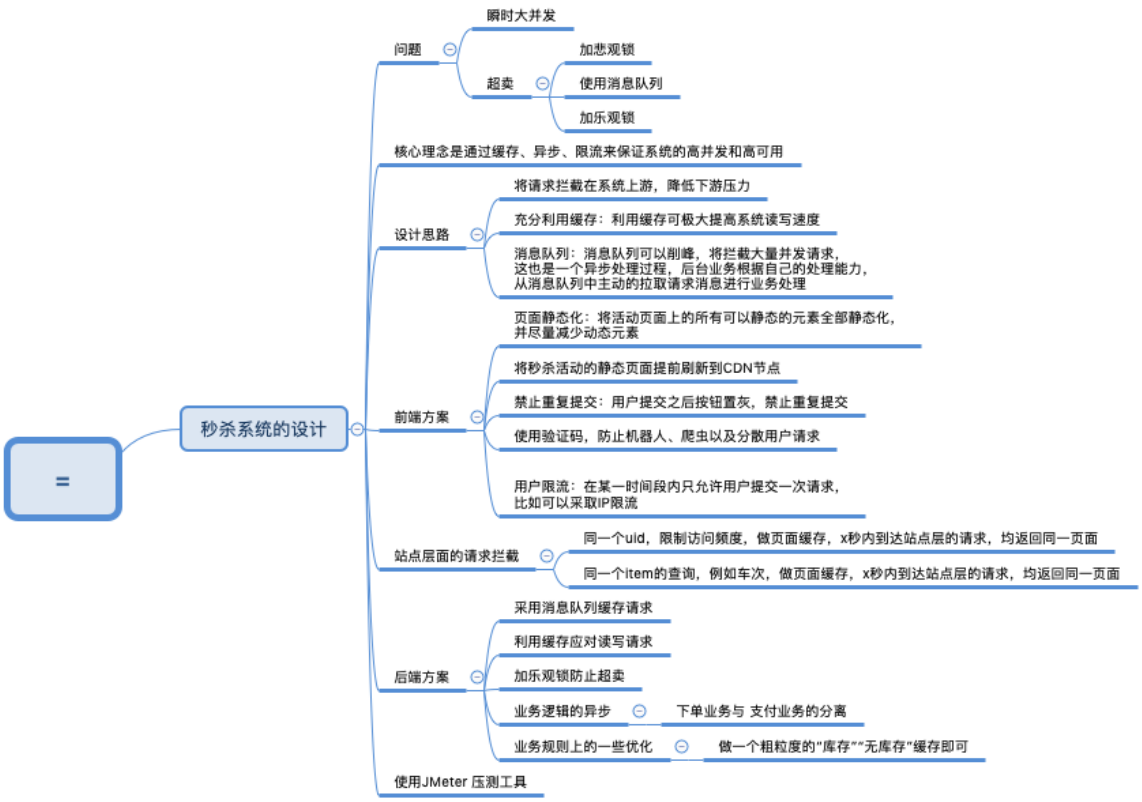
# 秒杀系统设计




## 解决思路

尽量将请求拦截在数据库的上游，因为一旦大量请求进入数据库，性能会急剧下降

## 架构和实现细节

- 前端模块（页面静态化、CDN、客户端缓存）
- 排队模块（Redis、队列实现异步下单）
- 服务模块（事务处理业务逻辑、避免并发问题）
- 防刷模块（验证码、限IP）

## 模块解析

### 前端模块

1. 页面静态化，将后台渲染模板的方式改成使用HTML文件与AJAX异步请求的方式，减少服务端渲染开销，同时将秒杀页面提前放到CDN
2. 客户端缓存，配置Cache-Control来让客户端缓存一定时间页面，提升用户体验
3. 静态资源优化，CSS/JS/图片压缩，提升用户体验

### 排队模块

1. 对Redis中的抢购对象预减库存，然后立即返回抢购成功请用户等待，这里利用了Redis将大部分请求拦截住，少部分流量进入下一阶段

2. 如果参与秒杀的商品太多，进入下一阶段的流量依然比较大，则需要使用消息队列，Redis过滤之后的请求直接放入到消息队列，让消息队列进行流量的第二次削峰

## 服务模块

1. 消息队列的消费者，业务逻辑是使用事务控制对数据库的下订单，减库存操作，且下订单操作要放到减库存操作之前，可以避免减库存update的行锁持有时间

## 防刷模块

1. 针对恶意用户写脚本去刷，在Redis中保存用户IP与商品ID进行限制
2. 针对普通用户疯狂的点击，使用JS控制抢购按钮，每几秒才能点击一次
3. 在后台生成数学计算型的验证码，使用Graphics、BufferedImage实现图片，ScriptEngineManager计算表达式

## 异常流程的处理

1. 如果在秒杀的过程中由于服务崩溃导致秒杀活动中断，那么没有好的办法，只能立即尝试恢复崩溃服务或者申请另寻时间重新进行秒杀活动
2. 如果在下订单的过程中由于用户的某些限制导致下单失败，那么应该回滚事务，立即告诉用户失败原因

# 难点+坑+复盘优化

## 难点

1. 理解整个架构设计的思路，围绕这个思路进行思考有什么方式可以做到，在开发过程中多进行压力测试反馈优化
2. 代码中异常情况的处理与业务上应急预案的准备

## 坑

1. 以上的解决方案能通过利用Redis与消息队列集群来承载非常高的并发量，但是运维成本高。比如Redis与消息队列都必须用到集群才能保证稳定性，会导致运维成本太高。所以需要有专业的运维团队维护。
2. 避免同一用户同时下多个订单，需要写好业务逻辑或在订单表中加上用户ID与商品ID的唯一索引；避免卖超问题，在更新数量的sql上需要加上>0条件

## 优化

1. 将7层负载均衡Nginx与4层负载均衡LVS一起使用进一步提高并发量
2. 以上是应用架构上的优化，在部署的Redis、消息队列、数据库、虚拟机偏向选择带宽与硬盘读写速度高的
3. 提前预热，将最新的静态资源同步更新到CDN的所有节点上，在Redis中提前加载好需要售卖的产品信息
4. 使用分布式限流减少Redis访问压力，在Nginx中配置并发连接数与速度限制，但如果有很多不同的活动同时进行则不适用