# 1.如何设计秒杀场景系统？

**秒杀业务的典型特点有：**  
1. 瞬时流量大   
2. 参与用户多，可秒杀商品数量少   
3. 请求读多写少   
4. 秒杀状态转换实时性要求高

**一次秒杀的流程可以分为三个阶段：**  
1. 活动未开始   
活动开始前，用户进入活动页，这个阶段有两种请求，一种是加载活动页信息，一个是查询活动状态得到未开始的结果， 一个用户进入页面两个请求各发起一次，这两种请求占比各半。   
2. 活动进行中   
这个阶段持续时间非常短，看到抢购按钮的用户大量发起秒杀请求，瞬时秒杀请求占比增高，能不能抗住秒杀请求就是秒杀系统是否能抗住高并发的关键。   
3. 活动结束   
当商品被抢购完，进入结束状态，请求情况同活动开始前。

### 设计思路：

**将请求拦截在系统上游，降低下游压力：**秒杀系统特点是并发量极大，但实际秒杀成功的请求数量却很少，所以如果不在前端拦截很可能造成数据库读写锁冲突，甚至导致死锁，最终请求超时。   
**充分利用缓存：**利用缓存可极大提高系统读写速度。   
**消息队列：**消息队列可以削峰，将拦截大量并发请求，这也是一个异步处理过程，后台业务根据自己的处理能力，从消息队列中主动的拉取请求消息进行业务处理。

## 1.前端的解决方案

主要是展示活动相关配置信息，活动背景图片，优惠力度，活动规则等相对静态的内容，通过web项目渲染成页面。

**页面静态化：**将活动页面上的所有可以静态的元素全部静态化，并尽量减少动态元素。通过CDN来抗峰值。   
**禁止重复提交：**用户提交之后按钮置灰，禁止重复提交。  
**用户限流：**在某一时间段内只允许用户提交一次请求，比如可以采取IP限流。

对于这样的请求，我们可以使用varnish反向代理，以页面相关的参数比如本次秒杀的活动ID和城市ID的hash为key把整个页面缓存在varnish机器上，而秒杀活动的状态等动态信息通过ajax来刷新。

## 2.后端的解决方案

**2.1服务端控制器层(网关层)**

**限制uid（UserID）访问频率：**我们上面拦截了浏览器访问的请求，但针对某些恶意攻击或其它插件，在服务端控制层需要针对同一个访问uid，限制访问频率。

**防止刷单神器**：有的用户为了抢到商品可能利用第三插件，去频繁的访问接口，这样给接口会带来很大的压力，为了避免这种刻意的刷单问题，可以在后台对同一个用户的访问频率做限制，可以预测接口的实际访问频率，然后对不同的接口，同一个用户做频率限制，如某个抢购入口，可以设置一个用户一分钟不能访问超过60次，10秒内不能超过20次等不同时段不同访问频率策略，这个可以通过redis等缓存框架做到，列入以某个接口+用户唯一性标准为key，可以选择value为string类型，如果缓存更多数据，value可以选择hash类型，value的值是int类型，这样用户每次访问接口，可以先判断该值有没有达到预设的访问频率限制的值，如果达到了，就告诉用户，你的访问太过频繁，请多长时间后再试，或者要求用户输入验证码（图片验证码或者短信验证码）。如果没有达到预设值，就给int值技术加一，在第一次访问的时候，即查询的前exist判断，如果不存在，插入第一条数据，并且设置过气时间，这个时间就是访问频率的时间限制例如一分钟只能访问60次，那么这里的过期时间就是60秒，预设值就是60次。这样就可以从一定程度上避免刷单问题。

**防止多个帐号刷单**：一个用户可能当初注册了很多的账号，平时不用，专门用来参加秒杀活动，这样其实也会造成系统压力。解决方式和上述差不多，只是对ip做限制，但是怎样做可能伤到真实的用户，因为有的场合如网吧，里面人购物其实是同一个ip出口，这个时候，其实也可以对用户的等级做限制，只有多高的等级才可以参加秒杀活动。

**2.2服务层**

**内存缓存**：参加秒杀系统的商品是事先可知的，可以将参加秒杀的商品信息事先缓存到redis等缓存系统中，这样可以大大的提高系统的吞吐量，减少关系型数据库的读写压力。

**削峰**：数据库削峰。对于秒杀系统瞬时会有大量用户涌入，所以在抢购一开始会有很高的瞬间峰值。对于关系型数据库而言，这个是致命的，是压垮系统很重要的原因，所以如何把瞬间的高流量变成一段时间平稳的流量也是设计秒杀系统很重要的思路。实现削峰的常用的方法有利用缓存和消息中间件等技术。

**异步处理**：秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式。

**可拓展**：服务的可扩展，可以水平添加机器将用户请求分担到不同的机器上去。数据库可扩展，支持分库分表，对于用户的请求，映射到不同的数据库，减少单台数据库的压力。

**2.3数据库层**

数据库层是最脆弱的一层，一般在应用设计时在上游就需要把请求拦截掉，数据库层只承担“能力范围内”的访问请求。所以，上面通过在服务层引入队列和缓存，让最底层的数据库高枕无忧。

案例：利用redis+mysql实现简单的秒杀系统

Redis是一个分布式key-value缓存系统，value支持多种数据结构，这里value可以选择两种类型，String（或者hash）：主要用于记录商品的库存，对商品减库存。Set集合（这里不要用list集合，list集合是可重复的，set是不可重复的，可以保证一个用户只卖一次，如果一个用户可以买多次那么可以使用list集合）：用于存储用户的id获取其他唯一确定一个用户的值。

在秒杀开始的前：可以使用批处理，将参加秒杀的产品信息缓存到redis中。这里将产品的业务唯一字段作为key，库存作为value。这里的key要和前端缓存的key一致。  
在秒杀开始时：：用户大量提交。根据用户提交的产品信息，获取到redis中需要的key值，查询缓存（为了保证缓存有效，如果第一次没有查询到，可以到数据库查询，然后在缓存一下，不过一般不会出现），得到库存量，判断当前库存是否大于零，如果大于零，判断当前的set集合中是否用该用户ID，如果没有，减库存并且将用户的ID放入集合中，并对库存减一，如果库存为0，提示用户，商品已售完等文案信息，如果集合中已经存在该用户id，则不做任何处理，直接处理下一个请求。直到最后库存售完，上面的过程可以利用redis事务和watch功能完成对数据一致性的控制即超卖问题。  
库存售完后：程序开始启动一个有个后台线程，可以阻塞等待商品库存售完的通知，在上面一步，库存一旦售完，后台进程获取set集合中的用户信息，异步处理需要操作的购买等后续操作。

## 详细解释：

秒杀状态就三种，未开始，可抢，已抢完，由两个因素共同决定   
1. 活动开始时间   
2. 剩余库存

**读取秒杀状态**的请求数并发也是非常高的，对于这个接口也要加上合适的缓存来处理。 对于活动开始时间，是一个较固定且不会发生变化的属性，并且，同时在线的秒杀活动数目并不多，所以把它也作为discount相关的信息，选择用响应快的**ehcache**来缓存。

**对于库存**，剩余库存个数，一般来说是全局需要一致的，可以用memcached来缓存，在秒杀的过程中，库存变化的非常快，如果直接对库存个数进行缓存，那么秒杀期间就需要频繁的更新缓存，像之前说的，虽然缓存是用来扛并发的，但要调用缓存的时机也要合理，memcached处理的并发请求越少，相对成功率就会越高。 其实对于秒杀活动来说，当时的剩余库存数在秒杀期间变化非常快，某个时间点上的库存个数并没有太大的意义，而用户更关心的是能不能抢，true or false。如果缓存true or false的话，这个值在秒杀期间是相对稳定的，只需要在库存耗尽的时候更新一次，而且为了防止这一次的更新失败，可以重复更新，利用memcached的cas操作，最后memcached也只会真正执行一次set写操作。 因为秒杀期间查询活动状态的请求都打在memcached上，减少写的频率可以明显减轻memcached的负担。

## 3.秒杀请求

### 3.1秒杀请求分析

秒杀请求是一个秒杀系统能不能抗住高并发的关键。因为秒杀请求和之前两个请求不同，它是写请求，不能缓存，而且是活动峰值的主力。

一个用户从发出秒杀请求到成功秒杀简单地说需要两个步骤： 1. 扣库存 2. 发送秒杀商品 这是至少两条数据库操作，而且扣库存的这一步，在mysql的innodb引擎行锁机制下，update的sql到了数据库就开始排队，期间数据库连接是被占用的，当请求足够多时就会造成数据库的拥堵。 可以看出，秒杀请求接口是一个耗时相对长的接口，而且并发越高耗时越长，所以首先，一定要限制能够真正进行秒杀的人数。

### 3.2秒杀队列校验

可抢状态需要第三个因素来决定，那就是当前秒杀的排队人数。加在判断库存剩余之前，挡上一层排队人数的校验， 即有库存 并且 排队人数 < 限制请求数 = 可抢，有库存 并且 排队人数 >= 限制请求数 = 抢完。

比如2500个名额秒杀名额，目标放过去3000个秒杀请求，那么排队人数记在哪里？ 这个可以有所选择，如果只记请求个数，可以用memcached的计数，一个用户进入秒杀流程increase一次，判断库存之前先判断队列长度，这样就限制了可参与秒杀的用户数量。

发起秒杀先去问排队队列是不是已满，满了直接秒杀失败，同时可以去更新之前缓存了是否可抢 true or false的缓存，直接把前台可抢的状态变为不可抢。没满继续查询库存等后续流程，开始扣库存的时候，把当前用户id入队。 这样，就限制了真正进入秒杀的人数。

这种方法，可能会有一个问题，既然限制了请求数，那就必须要保证放过去的用户能够秒完商品，假设有重复提交的用户，如果重复提交的量大，比如放过去的请求中有一半都是重复提交，就会造成最后没秒完的情况，怎么屏蔽重复用户呢？ 就要有个地方来记参与的用户id，可以使用redis的**set**结构来保存，这个时候set的size代表当前排队的用户数，扣库存之前add当前用户id到set，根据add是否成功的结果，来判断是否继续处理请求。

最终，把实际上几万个参与数据库操作的用户从减少到秒杀商品的级别，这是一个数据库可控制的范围，即使参与的用户再多，实际上也只处理了秒杀商品数量级的请求。

## 4.更多的优化

1.分库存。 一般这样做就已经能够满足常规秒杀的需求了，但有一个问题依然没有解决，那就是加锁扣库存依然很慢 。假设的活动秒杀的商品量能够再上一个量级，数据库也是扛不住这个量的，可以先把库存数放在redis上，然而单一库存加锁排队依然存在，库存这个热点数据会成为扣库存的瓶颈。一个解决的办法是分库存，比如总共有50000个秒杀名额，可以分50份，放在redis上的50个不同的key，那么每份上1000个库存，用户进入秒杀流程后随机到其中一个库存来修改，这样有50个库存数来竞争，缩短请求的排队时间。

## 5.总结

先用varnish挡掉了所有的读取状态请求 然后用ehcache缓存活动时间，挡掉活动未开始时查询活动状态的请求 memcached缓存是否可抢的状态，挡掉活动开始后到结束状态的活动查询请求 redis队列挡掉了活动进行中，过量的秒杀请求 到最后只留下了秒杀商品数量级的请求到数据库中。

# 2.如何解决超卖问题？

超发的原因：假设某个抢购场景中，我们一共只有100个商品，在最后一刻，我们已经消耗了99个商品，仅剩最后一个。这个时候，系统发来多个并发请求，这批请求读取到的商品余量都是99个，然后都通过了这一个余量判断，最终导致超发。

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。

有很多软件和服务都“乐观锁”功能的支持，例如Redis中的watch就是其中之一。通过这个实现，我们保证了数据的安全。