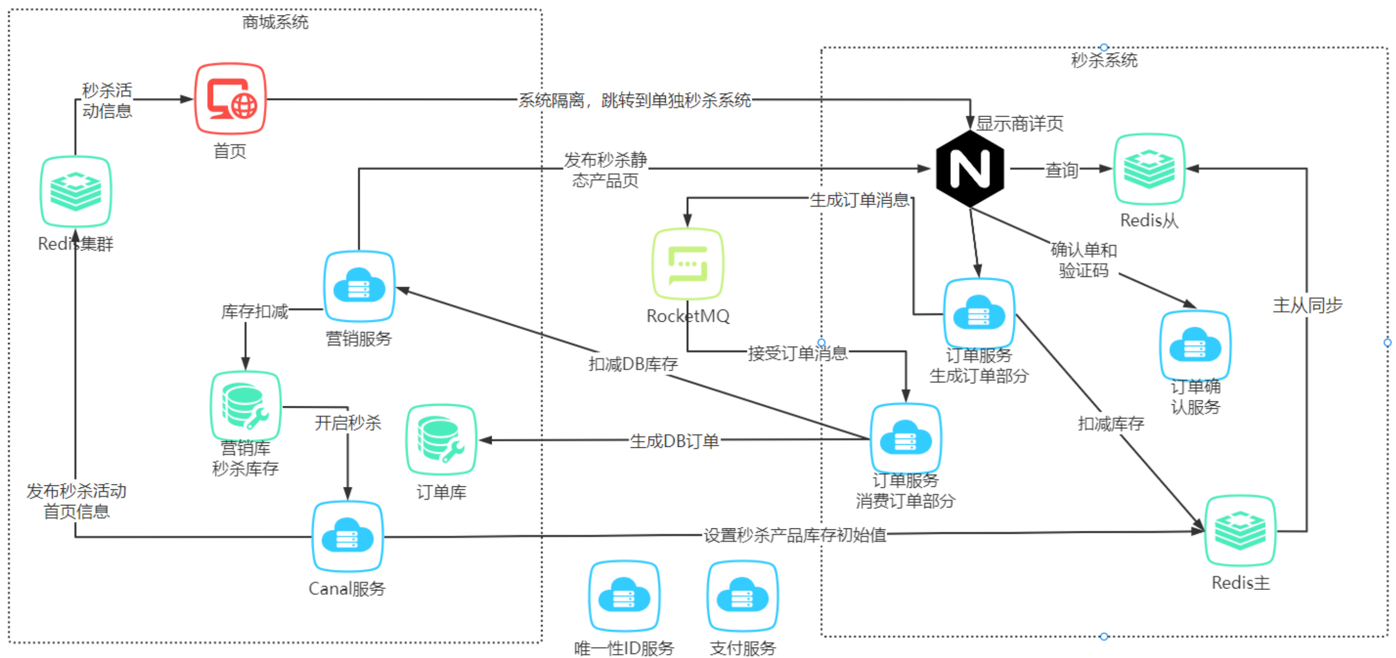
电商项目秒杀系统整体实现流程：
    秒杀业务流程梳理
秒杀前期流量管控
    为什么要前期流量管控
    预约系统设计
    预约系统优化
秒杀的事中流量管控
    削峰
        流量削峰
            验证码和问答题
            消息队列
    限流
        Nginx限流
        应用/服务层限流
            线程池限流
            API限流
            自定义限流
        分层过滤
限购、秒杀的库存与降级、热点
    限购
    库存扣减
        数据库方案
            行锁机制
            乐观锁
            数据库特性
    分布式锁方案
    高并发的扣减
        降级
            写服务降级
            商城库存扣减的实现
            读服务降级
            简化系统功能
    热点数据
        读热点
        写热点
防刷、风控和容灾处理
    防刷
    风控
    容灾

电商项目秒杀系统流量管控方案设计

电商项目秒杀系统整体实现流程：

我们的商城秒杀系统主要包含两个模块：tulingmall-sk-cart和tulingmall-sk-order。其中，tulingmall-sk-cart主要负责秒杀确认页和订单结算页的处理。tulingmall-sk-order负责秒杀订单处理。另外，doc目录则是配合openresty进行前端页面处理的相关文件。

后端完成后的部署架构如下图：



> 配合视频进行讲解

秒杀业务流程梳理

根据我们之前对秒杀业务的介绍，一场完整的秒杀活动的大概流程是这样的，我们可以结合上面的架构图一起梳理一下。

1. 运营人员在秒杀系统的运营后台，根据指定商品，创建秒杀活动，指定活动的开始时间、结束时间、活动库存等。

2. 活动开始之前，由秒杀系统运营后台开启秒杀，会同时往商城系统的Redis Cluster集群写入首页秒杀活动信息和往秒杀系统的Redis主从集群写诸如秒杀商品库存等信息。

3. 用户进入到秒杀商详页准备秒杀。

4. 商详页可以看到立即抢购的按钮，这里我们可以通过增加一些逻辑判断来限制按钮是否可以点击，比如是否设置了抢购用户等级限制，是否还有活动库存，是否设置了预约等等。如果都没限制，用户可以点击抢购按钮，进入到秒杀结算页。

5. 在结算页，用户可更改购买数量，切换地址、支付方式等，这里的结算元素也需要按实际业务来定，更复杂的场景还可以支持积分、优惠券、红包、配送时效等，并且这些都会影响最终价格的计算。

6. 确认无误后，用户提交订单，在这里后端服务可以调用风控、限购等接口，来完善校验，都通过之后，完成库存的扣减和订单的生成。

7.订单完成后，根据用户选择的支付方式跳转到对应的页面，比如在线支付就跳转到收银台，货到付款的话，就跳到下单成功提示页。

这样一来，秒杀业务从开始到用户抢购，到最后的活动结束关闭，整个流程就形成闭环了。当然上面列举的也只是主要的流程，实际业务可以在不同节点依据实际需求添加不同的业务功能，这个可以灵活调整。我们只列举主要要素。

秒杀前期流量管控

为什么要前期流量管控

如何有效地管控流量?

通过对秒杀流量的隔离，我们已经能够把巨大瞬时流量的影响范围控制在隔离的秒杀环境里了。接下来，我们开始考虑隔离环境的高可用问题，通俗点说，普通商品交易流程保住了，现在就要看怎么把秒杀系统搞稳定，来应对流量冲击，让秒杀系统也不出问题。方法很多，有流量控制、削峰、限流、缓存热点处理、扩容、熔断等一系列措施。

先来看流量控制。在库存有限的情况下，过多的用户参与实际上对电商平台的价值是边际递减的。举个例子，1万的荣耀手机，100万用户进来秒杀和1000万用户进来秒杀，对电商平台而言，所带来的经济效益、社会影响不会有10倍的差距。相反，用户越多，一方面消耗机器资源越多；另一方面，越多的人抢不到商品，电商平台的客诉和舆情压力也就越大。当然如果为了满足用户，让所有用户都能参与，秒杀系统也可以通过堆机器扩容来实现，但是成本太高，ROI不划算，所以我们需要，也可以提前对流量进行管控。

一般来说，很多电商平台，特别是头部电商很多时候会用“预约＋秒杀”作为主流营销玩法。

预约期内，开放用户预约，获取秒杀抢购资格，秒杀期内，具备抢购资格的用户真正开始秒杀。在预约期内，关键是锁定用户，这也是做前期流量管控的核心。

我们的商城系统虽然设计了相关的数据表sms_flash_promotion_log

对象 | sms_flash_promotion_log @tl_mall_p...

保存 | 添加字段 | 插入字段 | 删除字段 | 主键

字段 | 索引 | 外键 | 触发器 | 选项 | 注释 | SQL 预览

名	类型
id	int
member_id	int
product_id	bigint
member_phone	varchar
product_name	varchar
subscribe_time	datetime
send_time	datetime
gmt_create	datetime
gmt_modified	datetime

但是我们没有实现预约系统，所以光有这个表是不够的。不过我们可以看看如何来设计一个简单的预约系统。

预约系统设计

在进行系统设计之前，先看看预约需要的业务。

先从角色看，参与的有运营方，提供商品，进行预约活动的计划安排；终端用户，进行预约和秒杀行为；以及支撑预约活动的交易链路系统。

一般来说：

需要一个预约管理后台，进行活动的设置和关闭；

需要一个预约系统向预约过的用户发短信或消息提醒；

需要一个面向终端的预约核心微服务，提供给用户预约和取消预约能力；

商详在展示时获取预约信息的能力，比如当前商品是否预约，当前预约人数等等；

秒杀下单时检查用户预约资格的能力。

所以在数据库层面，对预约来讲，核心就是两个维度：预约活动和用户预约关系。所以需要两张表，一张是预约活动信息表，记录预约活动本身的信息，比如预约活动的开始结束时间，预约活动对应的秒杀活动信息，预约的商品信息等等；另一张是用户预约关系表，比如用户的ID，预约的活动ID，预约的商品等等。

预约系统优化

传统的预约模式，预约期是固定的时间段，用户在这个阶段内都可以预约;但在秒杀场景下，为了能够准确把控流量，控制预约人数上限，我们需要拓展预约期的定义，除了时间维度外，还要加入预约人数上限的维度，一旦达到上限，预约期就即时结束。

这实际上是给预约活动添加了一个自动熔断的功能，一旦活动太火爆，到达上限后系统自动关闭预约入口，提前进入等待秒杀状态。这样就可以准确把控人数，从而为秒杀期护航。

但是当用户都知道必须有预约才能参加秒杀时，用户就会在预约期抢占预约资格，那么此时的预约系统也具备一定程度秒杀系统的特点了。不过预约人数的把控不需要那么精确，只需要即时熔断即可，比如准备预约人数为100万，实际105万或者110万都没有什么问题。

对于头部电商平台，每次预约人数都可以达到千万量级的，因此为了更好的性能，往往还需要对数据库分库分表，主要是用户预约关系表。另外，对于预约历史数据，也需要有个定时任务进行结转归档，以减轻数据库的压力。

但是仅仅分库分表还是不够的，对高并发系统来说，要扛住大流量，肯定不能让流量击穿到数据库，所以需要设计缓存来抵挡。

首先是预约活动信息表，这是个很明显的读热点，所有的预约商品展示的时候都需要这份数据，很自然我们可以将数据在 Redis 缓存里存储，如果 Redis 缓存也扛不住，可以使用Redis 一主多从来扛，也可以使用服务的本地缓存。

对于用户预约关系表，是跟着用户走的，没有读热点问题，只要用户登录或者合适的时机将该用户的本次预约关系加载到Redis 缓存即可，在预约商品展示时从Redis 读取然后告诉用户是否已经预约。

用户进行预约的时候怎么办呢？虽然用户预约关系表可以做分库分表，本身又是个纯粹的insert操作，MySQL执行相对来说速度较快，但是要考虑某些热门商品会短时间挤入大量的用户，这个时候可以考虑使用消息中间件异步写入，做好消息的防重防丢失，同时前端提醒用户“预约排队中”。

另外，一般预约系统在业务设计上，需要在商详页展示当前预约人数给用户看，以营造商品火爆的气氛。我们自然就想到了可以在Redis 里记录一个预约人数的记录。商详页展示氛围的时候，会从Redis里获取到这个记录进行提示，而用户点击“立即预约”按钮进行预约时，会往这个key进行累加操作。

这个设计在预约流量没那么聚集时没什么问题，因为一般 Redis单片也能扛个七八万的OPS。而当预约期每秒中十几万，甚至几十万预约呢？显然这个Redis key就是典型的写热 key问题了。考虑到这个预约人数并不需要非常精确，这个热 key问题的解决我们可以考虑在本地缓存中累加，然后批量的方式写入Redis，比如累加了1000个人后一次性在Redis中incr 1000，这样就把对Redis的写压力降低了1000倍。

通过预约来控制流量属于事前管控，其实在事中，还有很多的手段来管控流量，我们来看看。

秒杀的事中流量管控

削峰

我们已经知道了秒杀有隔离和事前流量控制，其目的是降低流量的相互耦合和量级，减少对系统的冲击。秒杀系统事中流量管控——削峰和限流让系统更加稳健。

真实场景下的秒杀流量一般几秒内爬升到峰值，然后很快往平常值回归。我们现在需要做的就是通过削峰和限流，把这超大的瞬时流量平稳地承接下来，落到秒杀系统里。

削峰填谷概念一开始出现在电力行业，是调整用电负荷的一种措施，在互联网分布式高可用架构的演进过程中，也经常会采用类似的削峰填谷手段来构建稳定的系统。

削峰的方法有很多，可以分为无损和有损削峰。本质上，限流是一种有损技术削峰；而引入验证码、问答题以及异步化消息队列可以归为无损削峰，不过我们习惯上会把限流和削峰分开来说，所以我们这里也分开阐述。

流量削峰

我们已经知道秒杀的业务特点是库存少，最终能够抢到商品的人数取决于库存数量，而参与秒杀的人越多，并发数就越高，随之无效请求也就越多。

在秒杀开始的时刻，会出现巨大的瞬时流量，这个流量对资源的消耗也是巨大且瞬时的。

我们支撑秒杀系统的硬件资源是一定是有限的，它的处理能力也是恒定的，当有秒杀活动的时候，很容易繁忙导致请求处理不过来，而没有活动的时候，机器又是低负载运转。但是为了保证用户的秒杀体验，一般情况下我们的处理资源只能按照忙的时候来预估，这会导致资源的一个浪费。

因此我们需要设计一些规则，延缓并发请求，甚至过滤掉无效的请求，让真正可以下单的请求越少越好。总结来说，削峰的本质，一是让服务端处理变得更加平稳，二是节省服务器的机器成本。

互联网常用的削峰手段有哪些呢？我们一一来看看。

验证码和问答题

在秒杀交易流程中，引入验证码和问答题，有两个目的：一是快速拦截掉部分刷子流量，防止机器作弊，起到防刷的作用；二是平滑秒杀的毛刺请求，延缓并发，对流量进行削峰。

让用户在秒杀前输入验证码或者做问答题，不同用户的手速有快有慢，这就起到了让1s的瞬时流量平均到30s甚至1分钟的平滑流量中，这样就不需要堆积过多的机器应对1s的瞬时流量了。

在我们的商城系统里就使用验证码来进行削峰

小米8 全面屏游戏智能手机 12GB+64GB 黑色 全网通4G 双卡双待

AI智慧全面屏 6GB +64GB 亮黑色 全网通版 移动联通电信4G手机 双卡双待手机 双卡双待

2,699元

选择规格

金色16G	金色32G
银色16G	银色32G

□请选择规格

立即秒杀

库存数量 100

用户点击"立即秒杀"时需要从秒杀系统获取图片验证码，并进行渲染；用户手工输入验证码后，提交给秒杀系统进行验证码校验，如果通过就跳转至秒杀结算页。

小米8 全面屏游戏智能手机 12GB+64GB 黑色 全网通4G 双卡双待

AI智慧全面屏 6GB +64GB 亮黑色 全网通版 移动联通电信4G手机 双卡双待手机 双卡双待

2,699元

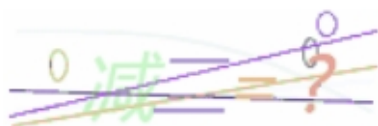
选择规格

金色16G	金色32G
银色16G	银色32G

□请选择规格

立即秒杀

库存数量 100



请输入验证码

在具体实现上，我们直接使用了HappyCaptcha生成验证码，tulingmall-sk-cart 秒杀确认单的CartItemController就有对应的实现

```xml
<!--HappyCaptcha-->
<dependency>
    <groupId>com.ramostear</groupId>
    <artifactId>Happy-Captcha</artifactId>
    <version>1.0.1</version>
</dependency>
```

```java
@ApiOperation("生成验证码-限流")
@GetMapping("/verifyCode")
public void generateImg(HttpServletRequest req, HttpServletResponse resp) throws I
    HappyCaptcha.require(req,resp)
            .style(CaptchaStyle.ANIM) //动画 or 图片
            .type(CaptchaType.ARITHMETIC_ZH) // 中文简体加、减、乘、除
            .build().finish();
}

Mark
@ApiOperation("检查验证码")
@RequestMapping(value = "/checkCode", method = RequestMethod.POST)
@ResponseBody
public CommonResult checkCode(@RequestParam String verifyCode,
                              HttpServletRequest request) throws IOException {
    if(!HappyCaptcha.verification(request, verifyCode, ignoreCase: true)){
        return CommonResult.failed("请填入正确的验证码");
    }else{
        return CommonResult.success( data: "验证通过");
    }
}
```

如果愿意，可以替换为其他成熟的验证码实现，这个看自己喜欢，不过在我们的验证码实现中，验证码放在了服务的本地session中，这个肯定是不合适的，从session共享角度来说，验证码应该放入Redis才是正确的，但是因为商城系统的登录中已经接入spring session解决验证码共享存储的问题，所以我们这里就没有重复实现了。

当然也可以生成验证码然后从session取出来自行放到redis即可，怎么获得HappyCaptcha生成的验证码呢? HappyCaptcha是放到session的，怎么从session中获得呢，看看HappyCaptcha.verification()就知道答案:

```java
public static boolean verification(HttpServletRequest request,String code,boolean ignoreCase){
    if(code == null || code.trim().equals("")){
        return false;
    }
    String captcha = (String)request.getSession().getAttribute(SESSION_KEY);
    return ignoreCase?code.equalsIgnoreCase(captcha):code.equals(captcha);
}
```

获得后然后再删除掉session中的验证码。

另外还需注意的是验证码放入主Redis后，如果选择从Nginx直接读取从Redis的方式，需要注意Redis主从同步的延迟问题，解决方案可以在Lua脚本中引入以下两者之一：

a.休眠后重试"os.execute("sleep " .. n)"

b.读从Redis未果，则读主Redis

当然，验证码本身也可以独立为一个微服务，因为当生成验证码本身成为性能瓶颈，可以验证码服务集群化或者预生成批量验证码并缓存，但是缓存的内容除了验证码的文字结果外，验证图片也要缓存。很多大厂就有独立的验证码服务集群，这个时候直接调用即可。

消息队列

除了验证码和问答题，另—种削峰方式是异步消息队列。

当服务A依赖服务B时，正常情况下服务A会直接通过RPC调用服务B的接口，当服务A调用的流量可控，且服务B的TP99和QPS能满足调用时，这是最简单直接的调用方式，没什么问题，目前大部分的微服务间调用也都是这样做的。

但是，试想一下，如果服务A的流量非常高(假设10万QPS)，远远大于服务B所能支持的能力(假设1万QPS)，那么服务B的CPU很快就会升高，TP99也随之变高，最终服务B被服务A的流量冲垮。

这个时候，消息队列就派上用场了，我们把一步调用的直接紧耦合方式，通过消息队列改造成两步异步调用，让超过服务B范围的流量，暂存在消息队列里，由B根据自己的服务能力来决定处理快慢，这就是通过消息队列进行调用解耦的常见手段。

常见的开源消息队列有Kafka、RocketMQ和RabbitMQ等，我们商城中大量使用了RocketMQ，秒杀中自然也使用了它。

限流

限流是系统自我保护的最直接手段，现实中的系统，总有所能承载的能力上限，一旦流量突破这个上限，就会引起实例宕机，进而发生系统雪崩，带来灾难性后果。

对于秒杀流程来说，从用户开始参与秒杀，到秒杀成功支付完成，实际上经历了很多的系统链路调用，中间有非常庞杂的系统在支撑，比如有商详、风控、登录、限购、购物车以及订单等很多交易系统。

那么对于秒杀的瞬时流量，如果不加筛选，不做限制，直接把流量传递给下游各个系统，对整个交易系统都是非常大的挑战，也是很大的资源浪费，所以主流的做法是从上游开始，对流量进行逐级限流，分层过滤，优质的有效的流量最终才能参与下单。

通过一系列的逐级限流、分层过滤，比如风控和防刷筛选刷子流量，通过限购和预约校验过滤无效流量，通过限流丢弃多余流量，最终秒杀系统给到下游的流量就是非常优质且少量的了。

限流常用的算法有令牌桶和漏桶，有关这两个算法的更多介绍，请大家自行查阅VIP的课程中专门讲述的章节。

Nginx限流

Nginx本身也提供了非常强大的限流功能，比如有两个专门的限流模块HttpLimitzone和HttpLimitReqest，HttpLimitzone用来限制一个客户端的并发连接数，HttpLimitReqest通过漏桶算法来限制用户的连接频率，我们用HttpLimitReqest来说明如何限流。

> ps：我们的OpenResty没有实现限流，是因为考虑到测试的需要，不是不能加相关配置。

HttpLimitReqest配置限流示例如下：

```
http {
  limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
  server {
    location /search/ {
    limit_req zone=one burst=2 nodelay;
    }
  }
}
```

limit_req_zone 是指令名称，也就是关键字，只能在http块中使用；

$binary_remote_addr 是Nginx内置绑定变量，比如$remote_port是客户端端口号；

zone=one:10m zone 是关键字，one是自定义的规则名称，后续代码中可以指定使用哪个规则；10m是指声明多大内存来支撑限流的功能，从理论上说一个1MB的区域可以包含大约16000个会话状态；

rate=1r/s rate是关键字，可以指定限流的阈值，r/s 意为每秒允许通过的请求数，我们这里就限定了每秒1请求。

再看两个实际例子：

```
limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
```

表示同一ip不同请求地址，进入名为one的zone，限制速率为5请求/秒。

```
limit_req_zone $binary_remote_addr $uri zone=two:10m rate=1r/s;
```

同一ip同一请求地址，进入名为two的zone，限制速率为1请求/秒。

limit_req 是指令名称，可在http，server， location块中使用，这个指令主要用于设置共享的内存zone和最大的突发请求大小；

zone=one 使用名为one的zone，这个zone之前使用limit_req_zone声明过；

burst=2 burst用于指定最大突发请求数，超过这个数目的请求会被延时；

nodelay 设置了nodelay，在突发请求数大于burst时，会立刻丢弃掉这部分请求，一般情况下会给客户端返回503状态。

在秒杀的场景下，一般会把 rate 和 burst 设置的很低，可以都为 1，即要求1个IP 1秒内只能访问1次。

这种设置一般对公司用户不太友好，公司内用户，他们的出口 IP 就那么几个，很容易就触发了限流，所以大家在参与头部电商的秒杀活动时，最好切换到 自己的手机网络，避免被“误杀”。

应用/服务层限流

以上是Nginx网关层的限流，接下来我们进入应用层的限流。应用层的限流手段也是比较多的，比如说线程池和API限流的方法。

线程池限流

Java原生的线程池原理相信你非常清楚，我们可以通过自定义线程池，配置最大连接数，以请求处理队列长度以及拒绝策略等参数来达到限流的目的。当处理队列满，而且最大线程都在处理时，多余的请求就会被拒绝策略丢弃，也就是被限流了。

API限流

上面介绍的线程池限流可以看做是一种并发数限流，对于并发数限流来说，实际上服务提供的QPS能力是和后端处理的响应时长有关系的，在并发数恒定的情况下，TP99越低，QPS就越高。

然而大部分情况是，我们希望根据QPS多少来进行限流，这时就不能用线程池策略了但是可以用Google提供的RateLimiter开源包，自己手写一个基于令牌桶的限流注解和实现，在业务API代码里使用。

当然了，现在大家用的Sentinel流量治理组件会比较多，可以从从流量路由、流量控制、流量整形、熔断降级、系统自适应过载保护、热点流量防护等多个维度来帮助保障微服务的稳定性，至于如何应用，后面的课程会继续详细讲到。

自定义限流

在前面章节中，我们曾经说到过订单重复下单问题，解决这个问题的思路是："在用户进入订单结算页面时，前端页面会先调用生成订单号的服务得到一个订单号，在用户提交订单的时候，在创建订单的请求中带着这个订单号"。

秒杀中为了避免重复订单，在秒杀订单结算页也做了类似的处理，但是可以想到，如果每个用户的请求都去申请一个订单号，在秒杀高并发的情况下是无法应对的，所以秒杀中做了改进

```
/*存放预制orderId的list，同时也可限流每秒允许个数，在更高并发的请求下，
 可以使用Disruptor代替 https://github.com/LMAX-Exchange/disruptor/wiki */
3 usages
private ConcurrentLinkedQueue<String> orderIdList = new ConcurrentLinkedQueue();
3 usages
private ConcurrentLinkedQueue<String> orderItemIdList = new ConcurrentLinkedQueue();
```

用一个线程安全的ConcurrentLinkedQueue预先存放一批订单ID，这样的话订单的ID无需去远程获取了。ConcurrentLinkedQueue中订单号的刷新则是通过定时任务刷新。

```
refreshService.scheduleAtFixedRate(new RefreshIdListTask(orderIdList,unqidFeignApi,orderItemIdList),
        initialDelay: 0, FETCH_PERIOD, TimeUnit.MILLISECONDS);
```

目前设定是100毫秒刷新一次，1秒钟最多从生成订单号的服务获得2000个订单ID，以常数的形式的写死在代码中的，这两个值其实可以写入配置中心进行热部署，方便秒杀根据实际情况来调整。

而从生成订单号的服务获得批量订单ID数，则是通过公式计算出来的，按照缺省值ConcurrentLinkedQueue每100毫秒最多有200个订单ID，这其实就起了一个限流的作用，因为在从ConcurrentLinkedQueue获得订单ID的时候，如果没有获取到，会直接返回中断用户的请求处理，返回一个处理失败。

```
public CommonResult generateConfirmMiaoShaOrder(Long productId
        , Long memberId, String token,Long flashPromotionId) throws BusinessException {

    String orderIdStr = orderIdList.poll();
    String orderItemIdStr = orderItemIdList.poll();
    if(null == orderIdStr || null ==orderItemIdStr){
        return CommonResult.failed("活动太火爆了，稍后再试试吧...");
    }
```

分层过滤

仔细考察秒杀的流量特征，比如某个秒杀商品1000个，秒杀时间为5分钟，现在有10万人来抢，2分钟内商品抢购完毕，那么后面3分钟其实商品已经无库存了。但是对后面3分钟的人发出的请求对于我们系统来说，其实是无效的请求，是没有必要把请求链路全部完成一遍的，这对资源其实是很大的浪费，所以我们可以在请求链路上层层过滤，把这部分无效请求提前筛选掉。所以在我们的秒杀实现中，到处可以看见相关的处理。

Nginx中，启用了本地缓存

```
# 共享字典，也就是本地缓存，名称叫做：stock_cache，大小1m
lua_shared_dict stock_cache 1m;
```

在stock.lua中则会检查本地缓存

```lua
-- 本地缓存的主要目的为库存检查，当商品的库存<=0时，提前终止秒杀
-- 这里从业务上来说，同样需要解决退单等引发的库存增加允许重新秒杀的情况，
-- 解决思路：同样可以订阅对应的Redis的channel，本次不做具体实现，Lua订阅Red
local item_cache = ngx.shared.stock_cache

-- 封装查询函数
function read_data(key, expire)
    -- 查询本地缓存
    local val = item_cache:get(key)
    if not val then
        ngx.log(ngx.ERR, "本地缓存查询失败，尝试查询Redis， key: ", key)
        -- 查询redis
        val = read_redis("127.0.0.1", 6379, key)
        -- 判断查询结果
        if not val then
            ngx.log(ngx.ERR, "redis查询失败，key: ", key)
            -- redis查询失败，给一个缺省值
            val = 0
        end
    end
    -- 查询成功，把数据写入本地缓存,expire秒后过期
    if tonumber(val) <= 0 then
        item_cache:set(key, val, expire)
    end
    -- 返回数据
    return val
```

与之相配合的，则是商详页中会根据这个返回值提示用户“秒杀商品已无库存，秒杀结束”，并关闭秒杀按钮。

```
function getProductStock(){
  console.log("productId:" + $("#productId").val()
  console.log("flashPromotionId:" + $("#flashPromo
  console.log("memberId:" + $("#memberId").val());
  $.get("cache/stock?productId="+$("#productId").
    console.log(data);
    if(data > 0){
      $("#secKillbtn").disabled=false;
      $("#flashPromotionCount").val(data);
    }else{
      console.log("秒杀商品已无库存，秒杀结束！");
      $("#secKillbtn").disabled=true;
    }
  })
}
```

在我们的服务层，不管是tulingmall-sk-cart秒杀确认单处理服务还是tulingmall-sk-order 秒杀订单处理服务都会对库存进行检查，比如SecKillConfirmOrderServiceImpl和SecKillOrderServiceImpl中都有下面的代码

```
private CommonResult confirmCheck(Long productId,Long memberId,String token)
    /*1、设置标记，如果售罄了在本地cache中设置为true*/
    Boolean localcache = cache.getCache( key: RedisKeyPrefixConst.MIAOSHA_STO
    if(localcache != null && localcache){
        return CommonResult.failed("商品已经售罄,请购买其它商品!");
    }
```

作用就是在实际下单和扣减库存前中断用户的请求链路的执行，起到一个层层过滤的作用。

限购、秒杀的库存与降级、热点

库存超卖，库存扣减热点的问题，它是秒杀系统面临的几大挑战之一。库存服务一般是商城平台的公共基础模块，负责所有商品可售卖数量的管理，对于库存服务来说，如果我只卖100件商品，那理想状态下，我希望外部系统就放过来100个下单请求就好了(以每单购买1件来说)，因为再多的请求过来，库存不足，也会返回失败。虽然我们的商城没有单独的库存服务，但是库存扣减操作和相关的数据表还是存在的，为了方便描述，我们下文还是统称为库存服务。

限购

并且对于像秒杀这种大流量、高并发的业务场景，更不适合直接将全部流量打到库存服务，所以这个时候就需要有个系统能够承接大流量，并且只放和商品库存相匹配的请求量到库存服务，而限购就能够承担这样的角色。限购之于库存，就像秒杀之于下单，前者都是后者的过滤网和保护伞。

顾名思义，限购的主要功能就是做商品的限制性购买。因为参加秒杀活动的商品都是爆品、稀缺品，所以为了让更多的用户参与进来，并让有限的投放量惠及到更多的人，所以往往会对商品的售卖做限制，一般限制的维度主要包括两方面。

商品维度限制：最基本的限制就是商品活动库存的限制，即每次参加秒杀活动的商品投放量。如果再细分，还可以支持针对不同地区做投放的场景，比如我只想在北京、上海、广州、深圳这些一线城市投放，那么就只有收货地址是这些城市的用户才能参与抢购，而且各地区库存量是隔离的，互不影响。

个人维度限制：就是以个人维度来做限制，这里不单单指同一用户ID，还会从同一手机号、同一收货地址、同一设备IP等维度来做限制。比如限制同一手机号每天只能下1单，每单只能购买1件，并且一个月内只能购买2件等。个人维度的限购，体现了秒杀的公平性。

有了这些功能支持之后，再做一个热门秒杀活动时，首先会在限购系统中配置活动库存以及各种个人维度的限购策略；然后在用户提单时，走下限购系统，通过限购的请求，再去做真实库存的扣减，这个时候可以减少到库存服务的量。

我们系统中就有对用户限购的约束检查，和前面的库存检查放在一个方法中，当然因为不是特别完善，所以目前没有实际启用。

那么在介绍完限购之后，下面我再来详细说一下上图中活动库存扣减的实现方案。

库存扣减

我们都知道，用户成功购买一个商品，对应的库存就要完成相应的扣减。而库存的扣减主要涉及到两个核心操作，一个是查询商品库存，另一个是在活动库存充足的情况下，做对应数量的扣减。两个操作拆分开来，都是非常简单的操作，但是在高并发场景下，不好的事情就发生了。

举个简单的例子，比如现在活动商品有2件库存，此时有两个并发请求过来，其中请求A要抢购1件，请求B要抢购2件，然后大家都去调用活动查询接口，发现库存都够，紧接着就都去调用对应的库存扣减接口，这个时候，两个都会扣减成功，但库存却变成了-1，也就是超卖了。

库存超卖的问题主要是由两个原因引起的，一个是查询和扣减不是原子操作，另一个是并发引起的请求无序。

所以要解决这个问题，我们就得做到库存扣减的原子性和有序性。该怎么去实现它呢?

数据库方案

行锁机制

利用数据库的行锁机制。这里有两种实现机制，

1、查询和扣减放在一个事务中，在查询库存的时候使用for update，事务结束行锁释放。

2、通过SQL语句，比如where语句的条件，保证库存不会被减到 0 以下，比如我们系统中StockManageServiceImpl锁定库存操作和扣减库存的操作都利用了这一点

```java
/*库存锁定,需要同时扣减商品库存和增加锁定库存，原来的实现：
        PmsSkuStock skuStock = skuStockMapper.selectByPrimaryKey(.......);
        skuStock.setLockStock(skuStock.getLockStock() + cartPromotionItem.getQuantity());
        skuStockMapper.updateByPrimaryKeySelective(skuStock);
 这里是先查再减，会有并发问题。
 * 实际扣减时也要判断商品库存是否足够扣减，否则会出现超卖*/
1 usage   USER-20221017CE\Administrator
@Override
public CommonResult lockStock(List<CartPromotionItem> cartPromotionItemList) {
    try {
        for (CartPromotionItem cartPromotionItem : cartPromotionItemList) {
            PmsSkuStockExample pmsSkuStockExample = new PmsSkuStockExample();
            pmsSkuStockExample.createCriteria()
                    .andIdEqualTo(cartPromotionItem.getProductSkuId())
                    .andStockGreaterThanOrEqualTo(cartPromotionItem.getQuantity());
            skuStockMapper.lockStockByExample(cartPromotionItem.getQuantity(),pmsSkuStockExample);
        }
        /*这里我们做了简单化处理，认为所有商品的库存锁定在业务上都可以成功，
         也就是商品库存一定足够扣减。
         实际要检查SQL操作返回行数，以供后续处理每个商品的锁定结果*/
        return CommonResult.success( data: true);
    }catch (Exception e) {
        log.error("锁定库存失败...{}",e);
        return CommonResult.failed();
    }
}
```

```xml
<update id="lockStockByExample" parameterType="java.lang.Integer">
  update pms_sku_stock
  set
  stock = stock - #{lockQuantity,jdbcType=INTEGER},
  lock_stock = lock_stock + #{lockQuantity,jdbcType=INTEGER}
  <if test="_parameter != null">
    <include refid="Update_By_Example_Where_Clause" />
  </if>
</update>
```

```java
/*订单支付后，实际扣减库存*/
1 usage   USER-20221017CE\Administrator
@Override
public CommonResult reduceStock(List<StockChanges> stockChangesList) {
    try {
            for (StockChanges changesProduct : stockChangesList) {
                PmsSkuStockExample pmsSkuStockExample = new PmsSkuStockEx
                pmsSkuStockExample.createCriteria()
                        .andIdEqualTo(changesProduct.getProductSkuId());
                skuStockMapper.reduceStockByExample(changesProduct.getCha
            }
        int result = skuStockMapper.updateSkuStock(stockChangesList);
        return CommonResult.success(result);
    }catch (Exception e) {
        log.error("订单支付后扣减库存失败...{}",e);
        return CommonResult.failed();
    }
}
```

```xml
<update id="updateSkuStock">
  UPDATE pms_sku_stock
  SET
  lock_stock = CASE id
  <foreach collection="itemList" item="item">
    WHEN #{item.productSkuId} THEN lock_stock - #{item.changesCount}
  </foreach>
  END
  WHERE
  id IN
  <foreach collection="itemList" item="item" separator="," open="(" close=")">
    #{item.productSkuId}
  </foreach>
```

乐观锁

每次查询库存的时候，除了库存值还有一个版本号，每次扣减库存时带上这个版本号进行扣减，比如：

select stock,verson from product where id = ?

update set stock = stock - ? ,version = version +1 where id = ? and version = ?

扣减失败，则需要重新查询，重新扣减。但会加重数据库的负担。

数据库特性

直接设置数据库的字段数据为无符号整数，这样减后库存字段值小于零时会直接执行 SQL 语句来报错。

总的来说，数据库方案简单安全，但是其性能比较差，无法适用于我们秒杀业务场景，在请求量比较小的业务场景下，是可以考虑的。

分布式锁方案

既然数据库不行，那能使用分布式锁吗？即通过 Redis 或者ZooKeeper来实现一个分布式锁，以商品维度来加锁，在获取到锁的线程中，按顺序去执行商品库存的查询和扣减，这样就同时实现了顺序性和原子性。

其实这个思路是可以的，只是不管通过哪种方式实现的分布式锁，都是有弊端的。以Redis的实现来说，仅仅在设置锁的有效期问题上，就让人头大。如果时间太短，那么业务程序还没有执行完，锁就自动释放了，这就失去了锁的作用；而如果时间偏长，一旦在释放锁的过程中出现异常，没能及时地释放，那么所有的业务线程都得阻塞等待直到锁自动失效，这与我们要实现高性能的秒杀系统是相悖的。所以通过分布式锁的方式可以实现，但不建议使用。

高并发的扣减

当秒杀活动开启，流量洪峰来临时，交易系统压力陡增，具体表现一般会包括CPU升高，IO 等待变长，请求响应时间TP99指标变差，整个系统变得越来越不稳定。为了力保核心交易流程，我们需要对非核心的一些服务进行降级，减轻系统负担，这种降级一般是有损的，属于“弃卒保帅”。

而秒杀的核心问题，是要解决单个商品的高并发读和高并发写的问题，这是典型的热点数据问题，我们需要有相应的机制，避免热点数据打垮系统。

降级

降级其实和削峰一样，降级解决的也是有限的机器资源和超大的流量需求之间的矛盾。如果你的资源够多，或者你的流量不够大，就不需要对系统进行降级了；只有当资源和流量的矛盾突出时，我们才需要考虑系统的降级。

降级一般是有损的，那么必然要有所牺牲，几种常见的降级:

写服务降级：牺牲数据一致性获取更高的性能；

读服务降级：故障场景下紧急降级快速止损。

我们来仔细分析下。

写服务降级

在多数据源（MySQL和Redis)的场景下，数据一致性一般是很难保证的。除非引入分布式事务，但分布式事务也会带来一些缺点，比如实现复杂、性能问题、可靠性问题等。因此一般在涉及金融资产类对一致性要求高的场景时，我们才会考虑分布式事务。

在流量不高的时候，我们的写请求可以直接先落入MySQL数据库，再通过监听数据库的Binlog变化，把数据更新进Redis 缓存，这种设计，缓存和数据库是最终一致的。通过缓存，我们可以扛更高流量的读操作，但是写操作仍然受制于数据库的磁盘IOPS，一般考虑一个数据库也就能支持3000~5000 TPS的写操作。

当流量激增的时候，我们就需要对以上的写路径进行降级，由同步写数据库降级成同步写缓存、异步写数据库，利用Redis 强大的OPS来扛流量，一般单个Redis 分片可达8~10万的OPS，Redis集群的OPS就更高了。

写请求首先直接写入Redis缓存，写入成功之后，发出写操作MQ（这一步可以放入另一个线程中操作），就可以返回客户端了。其他应用消费MQ，通过MQ异步化写数据库。

商城库存扣减的实现

回到我们的库存扣减上来，自然为了高并发，我们需要在Redis中进行内存扣减。在SecKillOrderServiceImpl中就是这样实现的，

```
private boolean preDecrRedisStock(Long productId) {
    Long stock = redisStockUtil.decr( key: RedisKeyPrefixConst.MIAOSHA_STOCK_CACHE_PREFIX + productId);
```

但是这样的实现有什么问题呢？这里根本没检查库存是否足够，是会导致超卖的。要知道，秒杀是一种促销活动，为了吸引更多的人气，更多的流量，是"赔本赚吆喝"，宁可少买，不可超卖！ 少买还可以再做一次"返场"活的，超卖肯定是不行的。

所以可以看到下面有检查是否超卖已经回滚库存的操作，但是这是必要的吗?

这段代码其实是从四期项目中直接继承过来的，位于四期代码的SecKillOrderServiceImpl中。

```
//还原库存
5 usages   fox
public void incrRedisStock(Long productId){
    if(redisOpsUtil.hasKey(RedisKeyPrefixConst.MIAOSHA_STOCK_CACHE_PREFIX + productId)){
        redisOpsUtil.incr( key: RedisKeyPrefixConst.MIAOSHA_STOCK_CACHE_PREFIX + productId);
    }
}
```

我们前面说过，要保证不超卖，查询和扣减需要是原子操作，正好Redis本身就是单线程的，天生就可以支持操作的顺序性，如果我们能在一次Redis 的执行中，同时包含查询和扣减两个命令就行。而且Redis可以执行Lua脚本的，并且可以保证脚本中的所有逻辑会在一次执行中按顺序完成。

所以正确的利用Redis扣减库存应该这么做

```
/* 还原缓存里的库存，主要是 当stock < 0时，有订单取消之类回滚库存的操作时，
 会导致增加的库存数量比实际的少，产生这个问题的主要原因是在扣减时未检查库存的数量，
 但是检查库存的数量，又容易导致库存超卖，库存超卖的问题主要是由两个原因引起的，
 一个是查询和扣减不是原子操作，另一个是并发引起的请求无序，
 所以解决这个问题可以采用执行Lua脚本的方法进行库存扣减，取消掉这个incrRedisStock操作：
 PO: Lua脚本参考如下，以一行注释一行代码形式呈现：
 -- -------------------Lua脚本代码开始****************************
 -- 调用Redis的get指令，查询活动库存，其中KEYS[1]为传入的参数1，即库存key
 local c_s = redis.call('get', KEYS[1])
 -- 判断活动库存是否充足，其中KEYS[2]为传入的参数2，即当前抢购数量
 if not c_s or tonumber(c_s) < tonumber(KEYS[2]) then
    return 0
 end
 -- 如果活动库存充足，则进行扣减操作。其中KEYS[2]为传入的参数2，即当前抢购数量
 redis.call('decrby',KEYS[1], KEYS[2])
 -- -------------------Lua脚本代码结束************************
 当然还可以将上面的脚本进行脚本预加载，预加载机制之一可以参考tulingmall-promotion中分布式锁的实现
 */
```

预加载可以有多种实现方式，一个是外部预加载好，生成了sha1然后配置到配置中心，这样Java代码从配置中心拉取最新sha1即可。另一种方式是在服务启动时，来完成脚本的预加载，并生成单机全局变量sha1

这里，我们通过Redis的高并发写能力，提升了系统性能，带来的牺牲就是缓存数据和数据库数据的一致性问题。为了追求高性能，牺牲一致性在大厂的设计中比较常见，对于异步造成的数据丢失等一致性问题，一般来说还会有定时任务一直在比对，以便最快发现问题，进行修复。

读服务降级

在做高可用系统设计时，要牢记就是微服务自身所依赖的外部中间件服务或者其他RPC服务，随时都可能发生故障，因此我们需要建设多级缓存，以便故障时能及时降级止损。

除了Redis缓存之外，还可以增加MongoDB或者ES缓存。当然了，你可以建立多个缓存副本，比如主Redis缓存外，再建立从Redis 缓存，这些都可以的，不过相应会增加资源成本和代码编写的复杂度。

假设当秒杀的Redis缓存出现故障时，我们就可以通过降级开关，快速将读请求降级到从Redis 缓存、MongoDB或者ES上。或者当Redis和备份缓存同时出现故障时(现实中很少出现同时故障的场景)，我们还是可以通过降级开关将流量切换到数据库上，让数据库暂时承压来完成读请求服务。

简化系统功能

简化系统功能就是指干掉一些不必要的流程，舍弃非核心功能

以京东或淘宝的商品详情页为例，上面除了商品的基本信息外，还有很多附加的信息，比如你是否收藏过该商品、商品的收藏总数量、商品的排行榜、评价和推荐等楼层。同样，对于秒杀结算页，还会有礼品卡、优惠券等虚拟支付路径。

如果是普通商品，这些附加信息当然是越多越好，一方面体现了系统的完整性，另一方面也可以多渠道引流促进转化。但是在秒杀场景下，这些信息是否有必要就需要视情况而定了，秒杀系统要求尽量简单，交互越少，数据越小，链路越短，离用户越近，响应就越快，因此非核心的功能在秒杀场景下都是可以降级的。

商城系统的商详页就采用了类似的做法，去除了普通商品详情页的很多信息，以加快商详页的显示，节约系统资源。

不过，实际运用中，这种非核心功能的有损降级，要视具体的SKU而定，一般为了降低影响范围，我们只对流量非常高的SKU进行降级。比如，如果是手机秒杀，一般是不需要降级的，但是像口罩这样的爆品，就需要针对SKU维度进行非核心功能的降级了。

降级开关的怎么设计呢，其实比较简单，核心思路就是通过配置中心，对降级开关进行变更，然后推送到各个微服务实例上。

热点数据

一般高并发的常规解决思路是：如果是数据库，可以通过分库分表来应对，如果是Redis，可以增加Redis集群的分片来解决，而应用层一般是无状态的设计。所以从数据库、Redis '缓存到应用服务，都是可以通过增加机器来水平扩展服务能力，解决高并发的问题。

然而，这样就能应对秒杀的挑战了吗？其实还不够，秒杀的核心问题是要解决单个商品的高并发读和高并发写问题，也就是要处理好热点数据问题。

所谓热点数据，是从单个数据被访问的频次角度去看的。单位时间（1s）内，一个数据非常频繁的被访问，就可以称之为热点数据，反之可以归为一般数据或冷数据。那么单位时间内究竟多高的频次才能称为热点数据呢?实际上并没有一个明确的定义，可以根据你自己的系统吞吐能力而定。

热点商品在进行秒杀时，只有这个SKU是热点，所以再怎么进行分库分表，或者增加Redis集群的分片数，热点商品SKU落在的那个分片的能力实际并没有提升，总会触达上限，把 Redis 打挂，最后可能引发缓存击穿、系统雪崩。那我们应该怎么解决这个棘手的热点问题呢?

我们把这个问题分为两类：读热点问题和写热点问题。下面我们分别展开讨论。

读热点

1，增加热点数据的副本数;

2，让热点数据离用户越近越好。

第一个解决方案，就是增加Redis从的副本数，然后业务层(Tomcat集群）轮询查询不同的副本，提高同一数据的QPS。一般情况下，单个Redis 从，可提供8~10万的查询，所以如果我们增加12个副本，就可以提供百万QPS的热点查询。

这个方法能解决热点问题，但成本比较高，如果你的集群分片数比较多，那分片数*副本数就是一笔不小的开销。

第二个解决方案，我们把热点数据再上移，在服务内部做热点数据的本地缓存，也就是让业务层的每个实例里都有份数据副本，读请求数据的时候，无需去 Redis 获取，直接从本地缓存里取。这时候，数据的副本数和服务一样多，另外请求链路减少了一层，而且也减少了对Redis 单片QPS上限的依赖，具有更高的可靠性和更高的性能。

这种方式热点数据的副本数随实例的增加而增加，非常容易扩展，扛高流量。但是本地缓存的数据延迟，业务要能够接受。其实在我们的首页里已经使用过这种方案了。

读热点还有一个比较简单粗暴的方法，那就是直接短路返回。这么说可能比较抽象，我举个例子，某个商品秒杀的时候，这个SKU是不支持使用优惠券的，那么优惠券系统在处理的时候，可以根据商品SKU编码，直接返回空的券列表，这样基本上不怎么耗资源，效率非常高。当然了，这种方式和具体商品的活动方式有关，不具有通用性，但是在几百万的流量面前，简单有效。

写热点

在前面流量管控的部分，我们说到点击“立即预约”的时候，会往“预约人数”这个Redis key 上进行累加操作，当几百万人同时预约的时候，这个key就是热点写操作了。

这个预约总人数有个特点，只是在前端给用户展示用，除此之外，没有其他用途，因此在并发的场景下，这个人数可以不用那么及时和精确，我们的思路就是先在JVM内存里累加，延迟提交到 Redis，这样就可以把 Redis 的OPS降低几十倍。

写热点还有一个场景就是库存的扣减，有一种思路，可以通过把一个热 key拆解成多个key的方式，避免热点问题。这种设计针对MySQL和Redis缓存都是适用的，但是涉及到对库存进行再细分，以及子库存挪动，非常复杂，而且边界问题比较多，容易出现库存不准的问题，需要谨慎小心的使用这种方法。

另一个思路就是对单SKU的库存直接在Redis 单分片上进行扣减，实际上，扣减库存在秒杀链路的末端，通过我们之前的削峰和限流的各种手段，真正到库存的流量是有限的，单片的Redis OPS 能承受得了。然后，我们可以针对单SKU的库存扣减进行单独限流，保证库存单片Redis的压力。这样双管齐下，单SKU的库存Redis 扣减压力就是可控的了。

当然高并发下的读写热点的处理方式还有很多种，我们会在后面的课程单列章节讲述大厂的常用方案。

防刷、风控和容灾处理

防刷

秒杀系统之所以流量高，主要是因为一般使用秒杀系统做活动的商品，基本都是稀缺商品。稀缺商品意味着在市场上具有较高的流通价值，那么它的这一特点，必定会引来一群“聪明”的用户，为了利益最大化，通过非正常手段来抢购商品，这种行为群体我们称之为黑产用户。

黑产用户总能想出五花八门的抢购方式，有借助物理工具，像“金手指”这种帮忙点击手机抢购按钮的；有通过第三方软件，按时准点帮忙触发App内的抢购按钮的；还有的是通过抓取并分析抢购的相关接口，然后自己通过程序来模拟抢购过程的。

可不管是哪种方式，其实都在做一件事，那就是先你一步。因为秒杀的抢购原则无外乎两种，要么是绝对公平的，即先到的请求先处理，暂时处理不了的，会把你放入到一个等待队列，然后慢慢处理。要么是非公平的，暂时处理不完的请求会立即拒绝，让你回到开始的地方，和大家一起再比谁先到，如此往复，直至商品售完。

因此黑产的方法也很简单，就是想法设法比别人快，发出的请求比别人多，就像在一个赛道上，给自己制造很多的分身，不仅保证自己比别人快，同时还要把别人挤出赛道，确保自己能够到达终点。

所以黑产对秒杀业务的威胁是巨大的，它不仅破坏了公平的抢购环境，而且给秒杀系统带来了庞大的性能开销，所以我们不能放任黑产流量对系统的肆意冲击，我们必须对抗它。既然黑产流量的特点是比正常流量快且频率高，那么我们也就可以从这两个方面来着手思考对策。

只针对第一个快的特点，其实在活动开始后，进来的流量我们都无法将其定义为非法流量，这个只能借助像风控这种多维度校验，才能将其识别出来，除非它跳步骤。而第二个高频率的特点，同时也是对秒杀系统造成危害最大的一种，我们还是有很多种手段来应对的。专门针对高频率以及跳步奏的非法手段常见的防刷方案有哪些呢?

Nginx有条件限流，是非常简单且直接的一种方式，这种方式可以有效解决黑产流量对单个接口的高频请求，但要想防止刷子不经过前置流程直接提单，还需要引入一个流程编排的Token机制。

Token机制，Token一般都是用来做鉴权的。放到秒杀的业务场景就是，对于有先后顺序的接口调用，我们要求进入下个接口之前，要在上个接口获得令牌，不然就认定为非法请求。同时这种方式也可以防止多端操作对数据的篡改，如果我们在 Nginx层做Token的生成与校验，可以做到对业务流程主数据的无侵入。

比如可以通过header_filter_by_lua_block，在返回的 header里增加流程Token。Token可以做MD5，加入商品编号、活动开始时间、自定义加密key等。

黑名单机制，黑名单机制分为本地黑名单和集群黑名单两种。该机制顾名思义，就是通过黑名单的方式来拦截非法请求的，但我们的核心问题是黑名单从哪里来呢?

总体来说，有两个来源：一个是从外部导入，可以是风控，也可以是别的渠道；而另一个就是自力更生，自己生成自己用。

比如前面介绍了Nginx有条件限流会过滤掉超过阈值的流量，但不能完全拦截，所以索性就不限流，直接全部放进来。然后我们自己实现一套“逮捕机制”，即利用Lua的共享缓存功能，去统计1秒内这个用户或者IP的请求频率，如果达到了我们设定的阈值，我们就认定其为黑产，然后将其放入到本地缓存黑名单。黑名单可以被所有接口共享，这样用户一旦被认定为黑产，其针对所有接口的请求，都将直接被全部拦截，实现刷子流量的0通过。

本地黑名单机制的优点就是简单、高效。但也正因为基于单机，如果黑产将请求频率控制在1*Nginx机器数以内，按请求理想散落的情况下，那么就不会被抓到，所以真要想通过频率来严格限制刷子请求，是可以借助Redis来实现集群黑名单的。

实现思路和单机的基本一致，就是使用的内存由本地变为了Redis，当然这也必然会影响接口的响应性能。

风控

风控在秒杀业务流程中非常重要，但风控的建立却是非常困难的。成熟的风控体系需要建立在大量的数据之上，并且要通过复杂的实际业务场景考验，不断地做智能修正，才能逐步提高风险识别的准确率。

像腾讯的风控，其依赖于庞大的微信、手Q生态体系的客户数据，日均调用量达2000亿次;京东的风控体系，涵盖零售、数科、物流、健康等线上线下多业务场景，跨多个领域且闭环；还有就是阿里的风控，相比京东，不仅有零售、数科、物流等，还有大文娱之类，场景更丰富。

那么为什么场景越丰富，相对来说风控的准确率越高呢?

这是因为风控的建设过程，其实就是一个不断完善用户画像的过程，而用户画像是建立风控的基础。一个用户画像的基础要素包括手机号、设备号、身份、IP、地址等，一些延展的信息还包括信贷记录、购物记录、履信记录、工作信息、社保信息等等。这些数据的收集，仅仅依靠单平台是无法做到的，这也是为什么风控的建立需要多平台、广业务、深覆盖，因为只有这样，才能够尽可能多地拿到用户数据。

有了这些数据，所谓的风控，其实就是针对某个用户，在不同的业务场景下，检查用户画像中的某些数据，是否触碰了红线，或者是某几项综合数据，是否触碰了红线。而有了完善的用户画像，黑产用户风控中的判定自然就越准。

容灾

机房容灾其实不仅仅是秒杀系统需要思考的，重要的软件系统，不管是互联网应用，还是传统应用，比如银行系统等，都需要考虑机房容灾的问题。不同的场景，容灾的设计也不尽相同，常见的互联网公司一般会怎么搭建容灾呢?

容灾，一般是指搭建多套(两套或以上)相同的系统，当其中一个系统出现故障时，其他系统能快速进行接管，从而持续提供7*24不间断业务。

在讨论容灾的时候，经常会听到“同城双活”“异地多活”等术语，它们都是不同的容灾方案，不同的方案，其技术要求、建设成本、运维成本都不一样。在多活架构下，对两套系统之间通信线路质量、时延要求很高，业内主流IT厂家比较认可的是单向时延2ms以内，超过这个时延，对“多活”的跨机房请求和数据同步的性能影响就会比较大。..

因此，涉及跨城市的多活，当城市距离较大时，比如上海和北京，那么这种物理上的时延很难克服。为了保证数据库的一致性，就需要付出很高的时间成本，往返几个来回时延叠加，RT就受不了了。所以异地多活单元化的设计其实非常复杂，成本高昂，即便是大厂也不一定能搭建好异地多活。

“同城双活”相对就简单一些，同城双活是在同城或相近区域内建立两个机房。同城双机房距离比较近，通信线路质量较好，比较容易实现数据的同步复制，保证高度的数据完整性和数据零丢失。

同城两个机房各承担一部分流量，一般入口流量完全随机，内部RPC调用尽量通过就近路由闭环在同机房，相当于两个机房镜像部署了两个独立集群，同城双活因为物理距离短，机房间的时延是有保证的。数据仍然是单点写到主机房数据库，然后实时同步到另外一个机房，读流量则完全可以做到机房内闭环。

双机房间的物理专线也必须是高可用的设计，至少需要两根以上进行互备，这样在专线故障时才有机会绕行避免不可用，这些在大厂里一般是运维团队在保障，我们稍微了解实现原理就可以。

> 有道云笔记链接：https://note.youdao.com/s/EQkM0J2h