秒杀系统核心交易链路优化 三

图灵：楼兰

今天要处理的问题：秒杀场景下如何进行限流。

今天要做的内容：

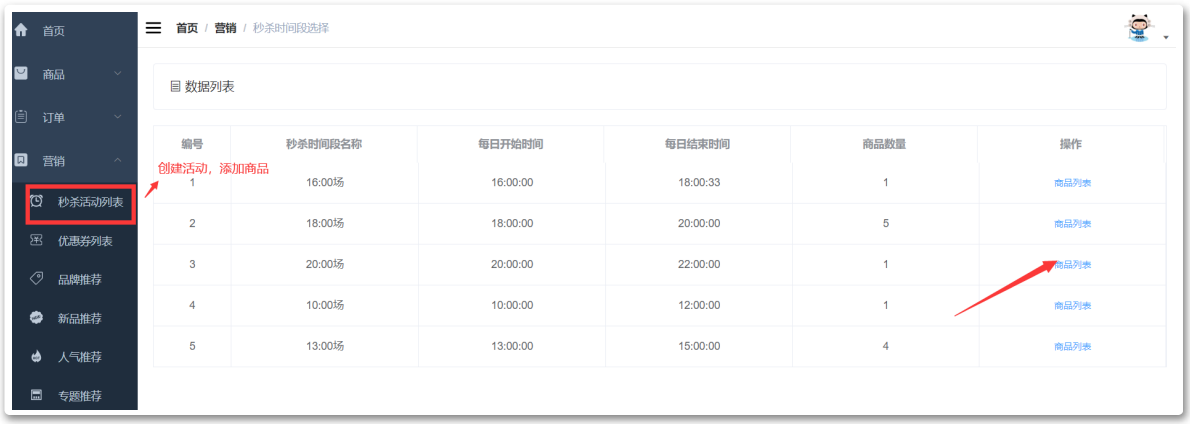
解决的问题：1、在秒杀页面，客户点击秒杀后，在前台弹出一个验证码，需要用户输入验证码才能往后端发送请求，这样能够错开秒杀下单的时间。

2、通过验证码，对后台下单请求进行保护，防止刷单，即绕开前端，直接往后端发送请求。

> 在秒杀页面开始秒杀后，客户点击秒杀按钮，要在前台弹出一个验证码，需要用户输入验证码才能往后端发请求，这样能够错开秒杀下单时间。
>
> 在我们的实现中，是要将memberId、producrId和验证码的值一起传入后台，后台返回一个token。然后再根据这个tokne拼接一个后台秒杀地址。这个token会存入到redis中。实际秒杀时，会增加一个判断，检测这个token是不是在redis中存在。如果不存在，就是机器刷单

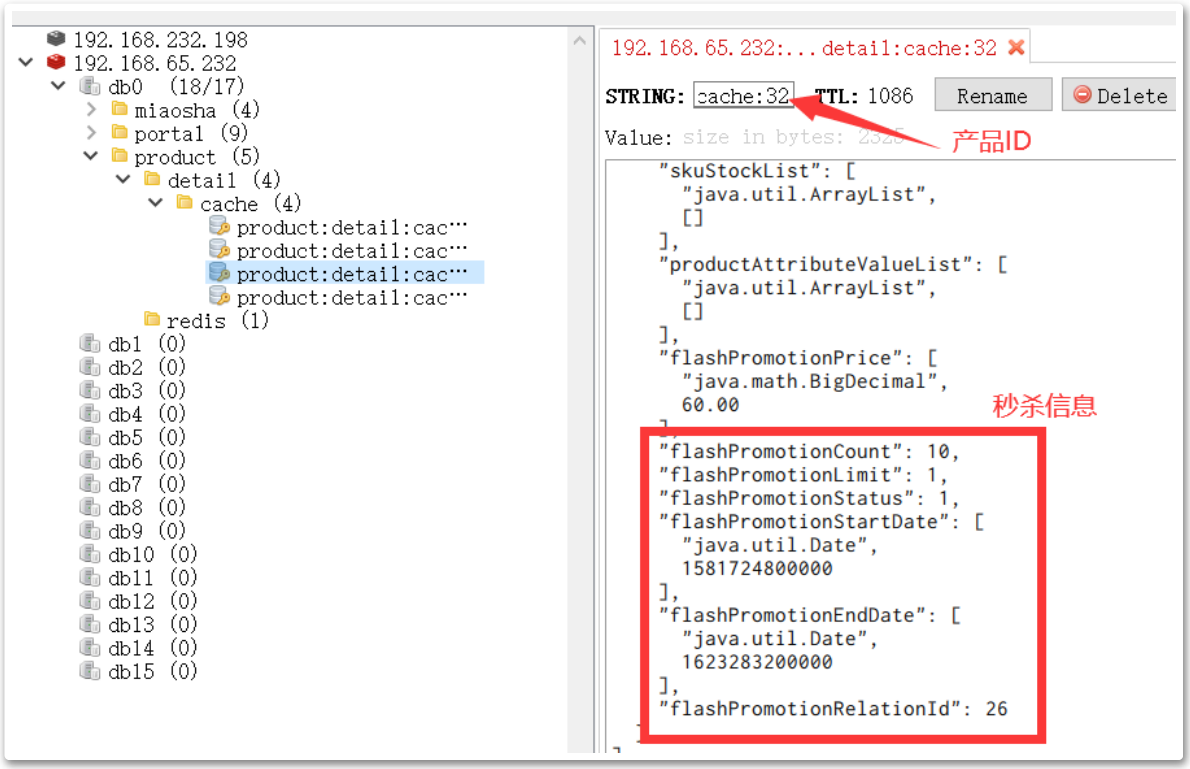
一、电商项目中秒杀的实现流程

1、在tmll-admin中添加秒杀活动，在秒杀活动中先设置活动的开始日期和结束日期，然后添加商品。



这个秒杀活动信息会保存到mysql中。sms_flash_promotion_product_relation表。

同时，添加商品后会将活动商品保存到ZK中。(路径/ZkLock/load_db/{productId})。然后，当访问到商城前端商品页时， http://localhost:8080/#/product/{productId}，会检查Redis中的产品信息缓存。如果Redis中没有产品信息，就会重建Redis缓存。key为product:detail:cache:{ProdId}



然后进入商城的单品页 http://localhost:8080/#/product/32 ， product.vue 那个"立即购买"的按钮就会变成"立即秒杀"

点击立即秒杀就会进入秒杀页。secKillDetail.vue

代码实现机制:

1、从Redis判断商品是否有秒杀活动。

> 一个商品要么就只能秒杀，要么就只能普通购买，这样是否合理？ 这就是为什么要单独独立出一套秒杀服务集群。

2、发送后台请求申请验证码。后台返回验证码图片，并将验证码的计算结果保存到Redis。

> 验证码的请求路径里header里的memeberId是怎么进去的。有什么用?
>
> 生成验证码图片的这个请求要怎么防刷?

3、保护后台请求接口。

输入验证码后，先验证输入的验证码结果，返回一个Token。这个Token会传入到接下来的商品确认页面，同时会保存到Redis当中，表示当前用户有购买秒杀商品的资格。有效期300秒，300秒内必须完成下单，否则就要重新申请秒杀资格。

在后续的下单过程中，需要传入这个Token才能正常下单。

> 验证码如果输入错误，是如何判断的?

二、如何加强限流方案的安全性

了解整理流程后，要继续深入思考下我们这个限流方案的安全性。

1> 针对验证码

> 针对验证码的安全性，可以加上之前的验证码内容。

1、我们做了这一套机制后，到底有多安全？ 下单请求依然是可以用机器人模拟的。

- 用户ID是存在Cookie当中的，可以拿到。
- 图形验证码是随机的，那就总有可能产生容易被机器识别的验证码。

2、怎么加强验证码本身的安全性

- 这个问题也是必须要前后台配合来思考的，而不是单独靠前端或者后端能够解决的。这个方案要如何设计？ 提高验证码安全性的措施：1、加干扰线或者干扰点，2，将关键字符变形并且在图形上串到一起。3、增加更多的前端交互，行为验证。
- 验证码的内容最好是一个比较复杂的题目，而不是简单的输入数字。这样可以有效延长下单请求的时长，更好的分散请求峰值。
- 图形验证码可以篡改。可以用PostMan另外访问生成图形验证码的接口，这时Redis里的值就被篡改了，不再是页面上看到的计算结果了。如何处理？1、增加更多的判断因素，例如IP。2、前端签名，后端验证签名。
- 输入了验证码之后，存在Redis中的验证码要及时删除。同时生成一个Token，代表当前用户有购买权限。这个Token有效期是非常短的。

> 针对验证机制的安全性，可以增加一些安全机制。

换一种验证码

我们动手来换一种复杂一点的验证码，HappyCaptcha 官网地址： https://gitee.com/ramostear/Happy-Captcha

换的方式比较简单，首先在pom.xml中加入HappyCaptcha的依赖

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

然后在OmsPortalOrderController中getVerifyCode方法，将生成验证码的部分修改一下：

```
try {
            //====== HappyCaptcha验证码 =======
            //这个步骤就会完成生成图片并且往response发送的步骤。
            HappyCaptcha.require(request,response).style(CaptchaStyle.ANIM)
                    .type(CaptchaType.ARITHMETIC_ZH)
                    .build().finish();
            Object captcha = request.getSession().getAttribute("happy-captcha"); //HappyCaptcha生成的验证码是String类型
            int code = Integer.parseInt(captcha.toString());
            log.info("验证码答案:{}",captcha);
            redisOpsUtil.set(RedisKeyPrefixConst.MIAOSHA_VERIFY_CODE_PREFIX + memberId + ":" + productId
                    ,code
                    ,300
                    ,TimeUnit.SECONDS);
            //====== HappyCaptcha验证码结束 =======
            return null;
        }catch(Exception e) {
            e.printStackTrace();
            return CommonResult.failed("秒杀失败");
        }
```

这样，前台的验证码就变成了一闪一闪的动画。并且是中文的加减法，更难破解。

可以看到整个HappyCaptcha的实现机制跟我们自己的实现机制是差不多的，也是使用session来存储答案。

其他还有哪些更难以破解的验证码？

2> 针对下单请求

我们的实现机制是要求将token拼凑到请求路径上来。这跟把token作为参数传递有什么区别？

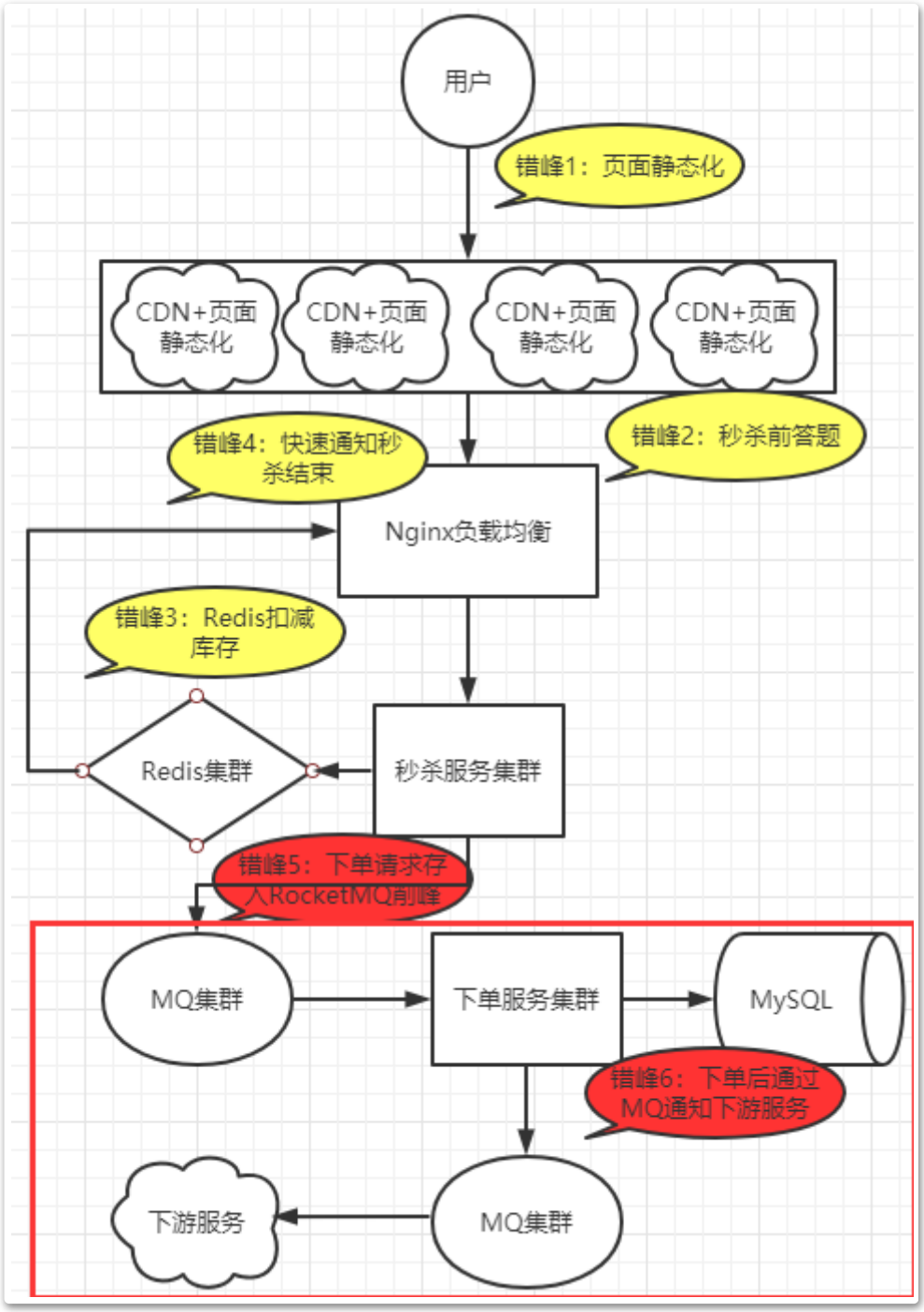
如果一个模拟程序需要使用机器来参与秒杀抢单，首先需要根据其他用户的请求来分析获取下单路径。如果是同一个请求路径，只是带的参数不同，那机器完全可以尝试用暴力破解的方式来尝试进行下单。如果碰巧传入了一个Redis中的token值，那他就下单成功了。但是现在把参数隐藏到了请求路径当中，动态的请求路径对于下单的机器来说，就比较难试探出请求的地址，这样就增加了他下单的难度。

三、电商整体的秒杀限流方案：

我们天天都在说三高，高并发、高可用、高可扩展，那到底应该如何去落地一个三高的设计方案？

构建大并发、高性能、高可用系统中几种通用的优化思路，可以抽象总结为“4 要 1 不要”原则。也就是：数据要尽量少、请求数要尽量少、路径要尽量短、依赖要尽量少，以及不要有单点。当然，这几点是你要努力的方向，具体操作时还是要密切结合实际的场景和具体条件来进行。

针对秒杀这个场景，其实方案设计往往比技术细节更为重要。因为你可以想象，每一个秒杀环节的经典问题，都意味着互联网的秒杀业务出现过大的问题，这都是实打实买来的教训。发现了问题之后才会有针对性的方案设计。那现在，我们整体来回顾下电商的秒杀限流方案。



错峰1：动静分离的本质是将包含浏览者信息的动态数据和不包含浏览者信息的静态资源区分开。例如在商品单品页，商品信息是不包含浏览者信息的，这部分就可以抽象出静态资源。而用户登录状态、cookie等这些动态数据也尽可能缓存起来，并且使缓存能够离用户更近。

错峰2：秒杀答题的形式可以是多种多样的，目的是防止机器刷单，以及错开用户的下单时长。在秒杀场景下，答题速度靠后的请求自然就没有库存了，也可以减少系统的请求量。

错峰3：缓存的作用主要有两个，一是快速扣减库存，保护数据库流量，并且库存扣减完成后，快速通知Nginx，屏蔽后续请求；二是提前识别热点数据，并且针对热点数据提供优化处理。处理的方案主要是三个，一是优化，二是限制，三是隔离，包括业务隔离、系统隔离、数据隔离。

错峰4：单独提供秒杀服务集群，有利于减少秒杀商品的超大流量对普通商品的性能冲击，不要让1%的商品影响到另外的99%。

后台错峰：这一部分是我们实战课程的重点。之前monkey老师带大家在后端针对秒杀场景做了非常多的设计与实战。我们这个图中每一个错峰点虽然在图上就是比较简单的一个点，但是深入进去，每个地方要考虑的细节都还是非常多的，大家可以回顾下之前的几节课，体会下如何在后端对秒杀服务做针对性的优化。

首先想到的是使用MQ进行削峰。但是实际上，后端需要考虑的三高问题也远不止MQ削峰这一步。每一个环节都需要考虑后端组件是否能够承载得住。例如秒杀服务集群，到底应该部署多大的集群？部署多少台机器呢？显然为了顶住秒杀的大流量，秒杀集群就需要部署得非常大。但是，如果在大部分没有秒杀服务的时间内，这个集群的资源就闲置得非常厉害。所以，虚拟化+云计算进行弹性部署也是非常重要的。在我们的项目实战课后面就会由诸葛老师给大家带来k8s和云部署的实战课程。

然后：在后端系统中，添加了Redis、MQ这样的一些中间产品。而这些产品集群本身，也存在效率低下、服务崩溃的风险。这样也就给系统整体带来了更多的风险点。那要怎么去屏蔽这些产品给系统带来的风险呢？大家可以思考一下，下一节课将会由fox老师给大家进行系统降级方面的设计。

题外话

方案优先 > 技术优先。学习技术的同时，都要增加对软件问题的思考，很多同学技术学得很快，但是缺乏思考。秒杀这种超大并发场景下的限流问题，不是任何一个技术或者任何一个步骤可以限制住的，需要一个完整全面的方案才能保证业务稳定性。所以我们在开发过程中，不能只埋头于技术点，要站在更高的角度，整体来理解解决方案，这样才能更深入的理解自己在做的事情，也才能真正来解决问题。这才是高级程序员与普通程序员真正的区别。

> 例如针对前端验证问题，还有哪些优化方案？
>
> 提前发Token。可以在秒杀前设置一个预约活动。 在活动中提前发放token。例如一个秒杀活动有20W个商品，那就可以预先准备200W个token。用户进行预约时，只发放200W个Token，其他人也能预约成功，但是其实没有获得token，那后面的秒杀，直接通过这个token就可以过滤掉一大部分人。相当于没有token的人都只预约了个寂寞。这也是互联网常用的一个套路。
>
> 例如针对超卖问题，在之前的课程中，介绍了如何使用Redis分布式锁防超卖。针对同一个商品ID，使用一把分布式锁，确实可以很快很方便的处理超卖问题。但是如果同时进行秒杀的商品多了呢？像京东、淘宝一场大型的秒杀活动，同时有成千上万个商品要进行秒杀，那就意味着同一时间Redis上锁解锁的操作会要执行成千上万次，这对Redis的性能消耗是相当巨大的，Redis就有可能升级成为新的性能瓶颈。这时该怎么办？
>
> 当然具体问题的解决方案从来不止一个，这里我们可以选择一种返璞归真的方案，把秒杀超卖的问题从分布式降级到本地JVM中，来获取极限性能。例如将秒杀服务接入配置中心，然后在秒杀服务开始前，由配置中心给每个应用服务实例下发一个库存数量。然后每次下单，每个服务器只管自己的库存数量，与其他应用服务器完全不进行库存同步，在各自的内存里扣减库存，这样就不会有超卖的情况发生。减少了网络消耗，性能也能够进一步提升。
>
> 这种方案可不可行呢？当然也会有一些问题需要去处理。有可能某给服务器上的库存很快消耗完了，而其他的服务器上仍有库存。整个服务就会表现为你抢不到商品，但是在你后面抢商品的人却能抢到商品。(你们在参与秒杀时有没有过这样的经历？)但是这在秒杀这种场景下，完全是可以接受的。另外，如果某一个应用服务器挂了，那给他分配的库存就

> 会丢失。这时候又要怎么办？其实也没必要再去设置什么复杂的逻辑，大不了少卖一点出去。反正都是售罄了，全卖完了，和卖了99%，其实没什么区别。这时只需要统计好订单的数量(可以通过MQ来统计，也可以通过Redis统计)，等秒杀活动的30分钟等待支付期过去后，再将没卖出去的库存重新丢回库存池，与没有付款而被取消的订单商品一起返场售卖就可以了。这也是很多互联网公司目前采用的方案。

最后虽然我们是后台开发工程师，但是前端也必须要有所了解。今天我们关注的这个问题，也不能只关注后端，需要前后端一起才能理解他的作用。