Web 大规模高并发请求和抢购的解决方案

电商的秒杀和抢购，对我们来说，都不是一个陌生的东西。然而，从技术的角度来说，这对于 Web 系统是一个巨大的考验。当一个 Web 系统，在一秒钟内收到数以万计甚至更多请求时，系统的优化和稳定至关重要。这次我们会关注秒杀和抢购的技术实现和优化，同时，从技术层面揭开，为什么我们总是不容易抢到火车票的原因？

一、大规模并发带来的挑战

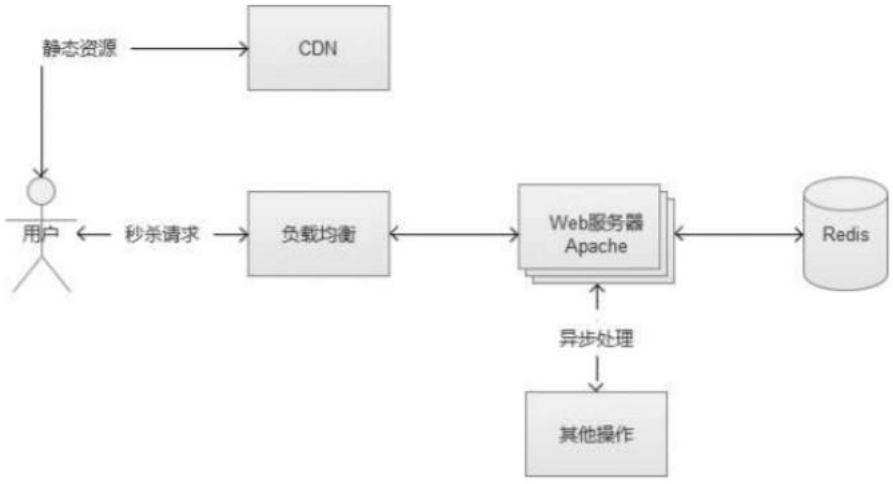
在过去的工作中，我曾经面对过 5w 每秒的高并发秒杀功能，在这个过程中，整个 Web 系统遇到了很多的问题和挑战。如果 Web 系统不做针对性的优化，会轻而易举地陷入到异常状态。我们现在一起来讨论下，优化的思路和方法哈。

1. 请求接口的合理设计

一个秒杀或者抢购页面，通常分为 2 个部分，一个是静态的 HTML 等内容，另一个就是参与秒杀的 Web 后台请求接口。

通常静态 HTML 等内容，是通过 CDN 的部署，一般压力不大，核心瓶颈实际上在后台请求接口上。这个后端接口，必须能够支持高并发请求，同时，非常重要的一点，必须尽可能“快”，在最短的时间里返回用户的请求结果。为了实现尽可能快这一点，接口的后端存储使用内存级别的操作会更好一点。仍然直接面向 MySQL 之类的存储是不合适的，如果有这种复杂业务的需求，都建议采用异步写入。

当然，也有一些秒杀和抢购采用“滞后反馈”，就是说秒杀当下不知道结果，一段时间后才可以从页面中看到用户是否秒杀成功。但是，这种属于“偷懒”行为，同时给用户的体验也不好，容易被用户认为是“暗箱操作”。



2. 高并发的挑战：一定要“快”

我们通常衡量一个 Web 系统的吞吐率的指标是 QPS（Query Per Second，每秒处理请求数），解决每秒数万次的高并发场景，这个指标非常关键。举个例子，我们假设处理一个业务请求平均响应时间为 100ms，同时，系统内有 20 台 Apache 的 Web 服务器，配置 MaxClients 为 500 个（表示 Apache 的最大连接数目）。

那么，我们的 Web 系统的理论峰值 QPS 为（理想化的计算方式）：

20*500/0.1 = 100000 （10 万 QPS）

咦？我们的系统似乎很强大，1 秒钟可以处理完 10 万的请求，5w/s 的秒杀似乎是“纸老虎”哈。实际情况，当然没有这么理想。在高并发的实际场景下，机器都处于高负载的状态，在这个时候平均响应时间会被大大增加。

就 Web 服务器而言，Apache 打开了越多的连接进程，CPU 需要处理的上下文切换也越多，额外增加了 CPU 的消耗，然后就直接导致平均响应时间增加。因此上述的 MaxClient 数目，要根据 CPU、内存等硬件因素综合考虑，绝对不是越多越好。可以通过 Apache 自带的 abench 来测试一下，取一个合适的值。然后，我们选择内存操作级别的存储的 Redis，在高并发的状态下，存储的响应时间至关重要。网络带宽虽然也是一个因素，不过，这种请求数据包一般比较小，一般很少成为请求的瓶颈。负载均衡成为系统瓶颈的情况比较少，在这里不做讨论哈。

那么问题来了，假设我们的系统，在 5w/s 的高并发状态下，平均响应时间从 100ms 变为 250ms（实际情况，甚至更多）：20*500/0.25 = 40000 （4 万 QPS）

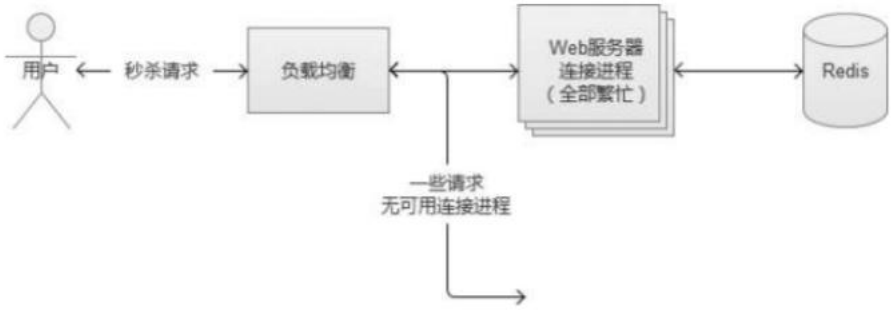
于是，我们的系统剩下了 4w 的 QPS，面对 5w 每秒的请求，中间相差了 1w。

然后，这才是真正的恶梦开始。举个例子，高速路口，1 秒钟来 5 部车，每秒通过 5 部车，高速路口运作正常。突然，这个路口 1 秒钟只能通过 4 部车，车流量仍然依旧，结果必定出现大塞车。（5 条车道忽然变成 4 条车道的感觉）

同理，某一个秒内，20*500 个可用连接进程都在满负荷工作中，却仍然有 1 万个新来请求，没有连接进程可用，系统陷入到异常状态也是预期之内。

其实在正常的非高并发的业务场景中，也有类似的情况出现，某个业务请求接口出现问题，响应时间极慢，将整个 Web 请求响应时间拉得很长，逐渐将 Web 服务器的可用连接数占满，其他正常的业务请求，无连接进程可用。

更可怕的问题是，是用户的行为特点，系统越是不可用，用户的点击越频繁，恶性循环最终导致“雪崩”（其中一台 Web 机器挂了，导致流量分散到其他正常工作的机器上，再导致正常的机器也挂，然后恶性循环），将整个 Web 系统拖垮。



3. 重启与过载保护

如果系统发生“雪崩”，贸然重启服务，是无法解决问题的。最常见的现象是，启动起来后，立刻挂掉。这个时候，最好在入口层将流量拒绝，然后再将重启。如果是 redis/memcache

这种服务也挂了，重启的时候需要注意“预热”，并且很可能需要比较长的时间。

秒杀和抢购的场景，流量往往是超乎我们系统的准备和想象的。这个时候，过载保护是必要的。如果检测到系统满负载状态，拒绝请求也是一种保护措施。在前端设置过滤是最简单的方式，但是，这种做法是被用户“千夫所指”的行为。更合适一点的是，将过载保护设置在 CGI 入口层，快速将客户的直接请求返回。

二、作弊的手段：进攻与防守

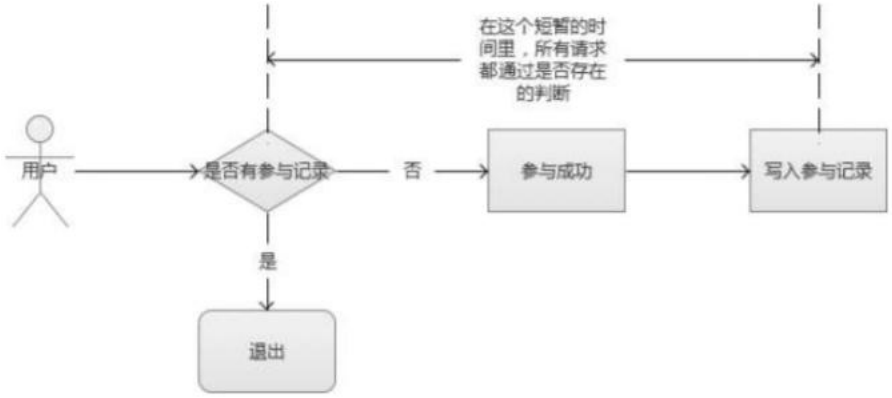
秒杀和抢购收到了“海量”的请求，实际上里面的水分是很大的。不少用户，为了“抢“到商品，会使用“刷票工具”等类型的辅助工具，帮助他们发送尽可能多的请求到服务器。还有一部分高级用户，制作强大的自动请求脚本。这种做法的理由也很简单，就是在参与秒杀和抢购的请求中，自己的请求数目占比越多，成功的概率越高。

这些都是属于“作弊的手段”，不过，有“进攻”就有“防守”，这是一场没有硝烟的战斗哈。

1. 同一个账号，一次性发出多个请求

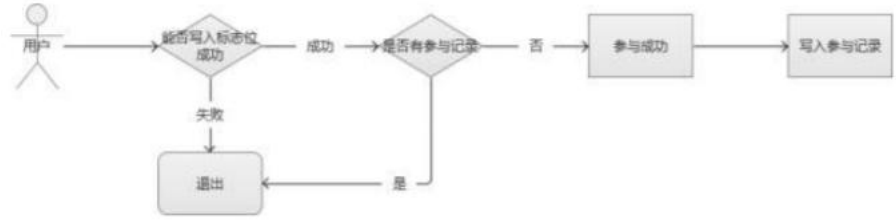
部分用户通过浏览器的插件或者其他工具，在秒杀开始的时间里，以自己的账号，一次发送上百甚至更多的请求。实际上，这样的用户破坏了秒杀和抢购的公平性。

这种请求在某些没有做数据安全处理的系统里，也可能造成另外一种破坏，导致某些判断条件被绕过。例如一个简单的领取逻辑，先判断用户是否有参与记录，如果没有则领取成功，最后写入到参与记录中。这是个非常简单的逻辑，但是，在高并发的场景下，存在深深的漏洞。多个并发请求通过负载均衡服务器，分配到内网的多台 Web 服务器，它们首先向存储发送查询请求，然后，在某个请求成功写入参与记录的时间差内，其他的请求获查询到的结果都是“没有参与记录”。这里，就存在逻辑判断被绕过的风险。



应对方案：

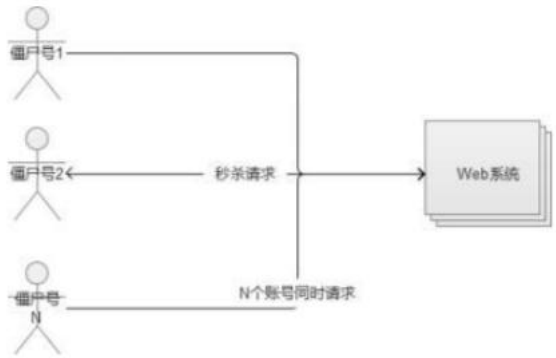
在程序入口处，一个账号只允许接受 1 个请求，其他请求过滤。不仅解决了同一个账号，发送 N 个请求的问题，还保证了后续的逻辑流程的安全。实现方案，可以通过 Redis 这种内存缓存服务，写入一个标志位（只允许 1 个请求写成功，结合 watch 的乐观锁的特性），成功写入的则可以继续参加。



或者，自己实现一个服务，将同一个账号的请求放入一个队列中，处理完一个，再处理下一个。

2. 多个账号，一次性发送多个请求

很多公司的账号注册功能，在发展早期几乎是没有限制的，很容易就可以注册很多个账号。因此，也导致了出现了一些特殊的工作室，通过编写自动注册脚本，积累了一大批“僵尸账号”，数量庞大，几万甚至几十万的账号不等，专门做各种刷的行为（这就是微博中的“僵尸粉“的来源）。举个例子，例如微博中有转发抽奖的活动，如果我们使用几万个“僵尸号”去混进去转发，这样就可以大大提升我们中奖的概率。



这种账号，使用在秒杀和抢购里，也是同一个道理。例如，iPhone 官网的抢购，火车票黄牛党。

应对方案：

这种场景，可以通过检测指定机器 IP 请求频率就可以解决，如果发现某个 IP 请求频率很高，可以给它弹出一个验证码或者直接禁止它的请求：

弹出验证码，最核心的追求，就是分辨出真实用户。因此，大家可能经常发现，网站弹出的验证码，有些是“鬼神乱舞”的样子，有时让我们根本无法看清。他们这样做的原因，其实也是为了让验证码的图片不被轻易识别，因为强大的“自动脚本”可以通过图片识别里面的字符，然后让脚本自动填写验证码。实际上，有一些非常创新的验证码，效果会比较好，例如给你一个简单问题让你回答，或者让你完成某些简单操作（例如百度贴吧的验证码）。直接禁止 IP，实际上是有些粗暴的，因为有些真实用户的网络场景恰好是同一出口 IP 的，可能会有“误伤“。但是这一个做法简单高效，根据实际场景使用可以获得很好的效果。

3. 多个账号，不同 IP 发送不同请求

所谓道高一尺，魔高一丈。有进攻，就会有防守，永不休止。这些“工作室”，发现你对单机 IP 请求频率有控制之后，他们也针对这种场景，想出了他们的“新进攻方案”，就是不断改变 IP。

有同学会好奇，这些随机 IP 服务怎么来的。有一些是某些机构自己占据一批独立 IP，然后做成一个随机代理 IP 的服务，有偿提供给这些“工作室”使用。还有一些更为黑暗一点的，就是通过木马黑掉普通用户的电脑，这个木马也不破坏用户电脑的正常运作，只做一件事情，就是转发 IP 包，普通用户的电脑被变成了 IP 代理出口。通过这种做法，黑客就拿到了大量的独立 IP，然后搭建为随机 IP 服务，就是为了挣钱。

应对方案：

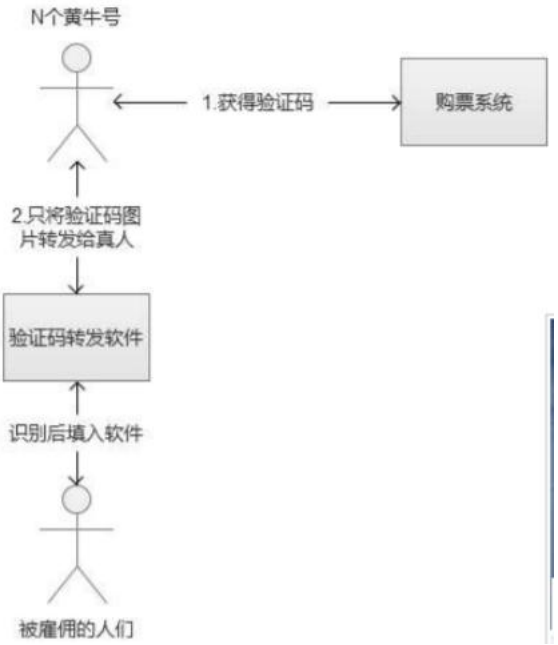
说实话，这种场景下的请求，和真实用户的行为，已经基本相同了，想做分辨很困难。再做进一步的限制很容易“误伤“真实用户，这个时候，通常只能通过设置业务门槛高来限制这种请求了，或者通过账号行为的”数据挖掘“来提前清理掉它们。

僵尸账号也还是有一些共同特征的，例如账号很可能属于同一个号码段甚至是连号的，活跃度不高，等级低，资料不全等等。根据这些特点，适当设置参与门槛，例如限制参与秒杀的账号等级。通过这些业务手段，也是可以过滤掉一些僵尸号。

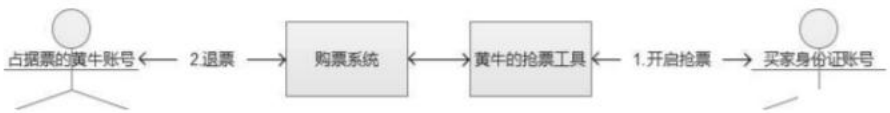
4. 火车票的抢购

看到这里，同学们是否明白你为什么抢不到火车票？如果你只是老老实实地去抢票，真的很难。通过多账号的方式，火车票的黄牛将很多车票的名额占据，部分强大的黄牛，在处理验证码方面，更是“技高一筹“。

高级的黄牛刷票时，在识别验证码的时候使用真实的人，中间搭建一个展示验证码图片的中转软件服务，真人浏览图片并填写下真实验证码，返回给中转软件。对于这种方式，验证码的保护限制作用被废除了，目前也没有很好的解决方案。



因为火车票是根据身份证实名制的，这里还有一个火车票的转让操作方式。大致的操作方式，是先用买家的身份证开启一个抢票工具，持续发送请求，黄牛账号选择退票，然后黄牛买家成功通过自己的身份证购票成功。当一列车厢没有票了的时候，是没有很多人盯着看的，况且黄牛们的抢票工具也很强大，即使让我们看见有退票，我们也不一定能抢得过他们哈。



最终，黄牛顺利将火车票转移到买家的身份证下。

解决方案：

并没有很好的解决方案，唯一可以动心思的也许是对账号数据进行“数据挖掘”，这些黄牛账号也是有一些共同特征的，例如经常抢票和退票，节假日异常活跃等等。将它们分析

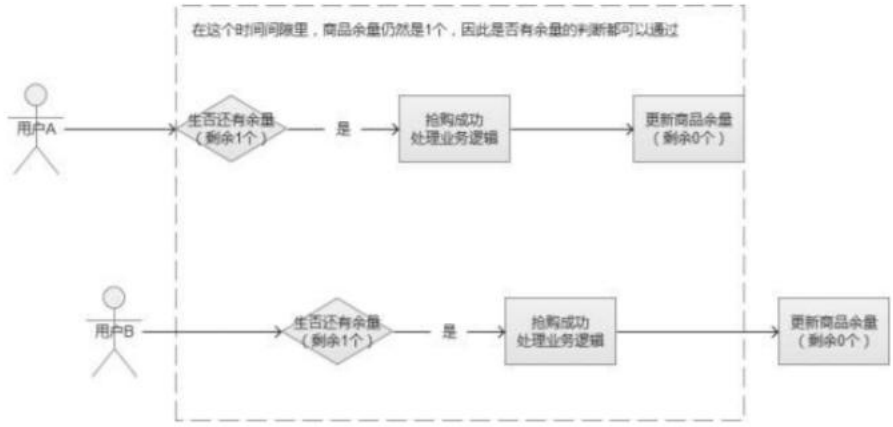
出来，再做进一步处理和甄别。

三、高并发下的数据安全

我们知道在多线程写入同一个文件的时候，会存现“线程安全”的问题（多个线程同时运行同一段代码，如果每次运行结果和单线程运行的结果是一样的，结果和预期相同，就是线程安全的）。如果是 MySQL 数据库，可以使用它自带的锁机制很好的解决问题，但是，在大规模并发的场景中，是不推荐使用 MySQL 的。秒杀和抢购的场景中，还有另外一个问题，就是“超发”，如果在这方面控制不慎，会产生发送过多的情况。我们也曾经听说过，某些电商搞抢购活动，买家成功拍下后，商家却不承认订单有效，拒绝发货。这里的问题，也许并不一定是商家奸诈，而是系统技术层面存在超发风险导致的。

1. 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。（同文章前面说的场景）

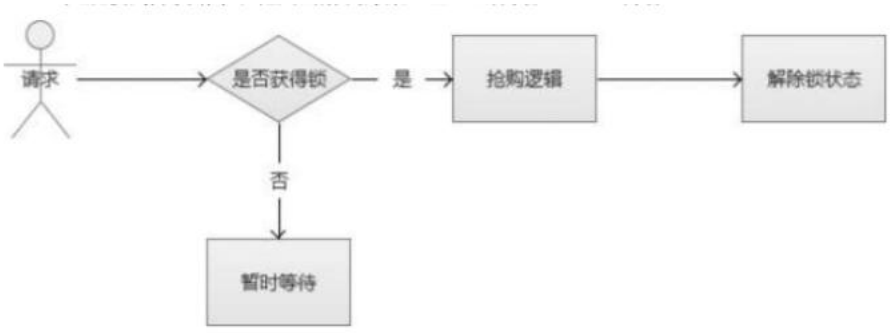


在上面的这个图中，就导致了并发用户 B 也“抢购成功”，多让一个人获得了商品。这种场景，在高并发的情况下非常容易出现。

2. 悲观锁思路

解决线程安全的思路很多，可以从“悲观锁”的方向开始讨论。

悲观锁，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待。



虽然上述的方案的确解决了线程安全的问题，但是，别忘记，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有

机会抢到这个“锁”，这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗尽，系统陷入异常。

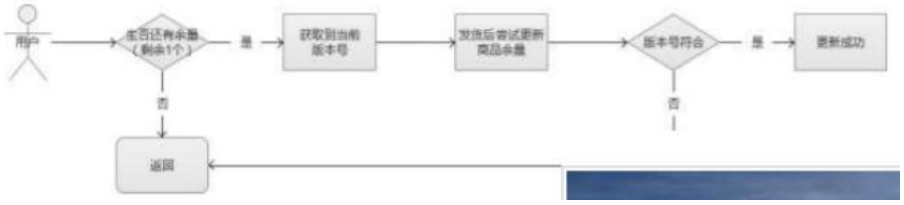
3. FIFO 队列思路

那好，那么我们稍微修改一下上面的场景，我们直接将请求放入队列中的，采用 FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程的感觉哈。

然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终 Web 系统平均响应时候还是会大幅下降，系统还是陷入异常。

4. 乐观锁思路

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。

四、小结

互联网正在高速发展，使用互联网服务的用户越多，高并发的场景也变得越来越多。电商秒杀和抢购，是两个比较典型的互联网高并发场景。虽然我们解决问题的具体技术方案可能千差万别，但是遇到的挑战却是相似的，因此解决问题的思路也异曲同工。