高并发秒杀系统的设计与实现

秒杀系统分析

秒杀系统的三个问题

问题一：为什么需要秒杀系统？

通俗点讲，电商平台的本质是在线上撮合买卖双方的购销需求，达成交易。虽然是线上交易，但也遵守朴素的经济学原理，线下的商场为了促进销售一般会采用各种促销让利的方式，吸引比平常更多的消费者购买，常见的促销方式有单品满减、总价优惠、赠品、会员优惠等。有时候很多的商品甚至是亏损出售，就是为了吸引更多的人气，更多的流量，所谓“赔本赚吆喝”，线上交易自然也是如此，秒杀就是为了这个目的。

问题二：京东、阿里巴巴等头部电商平台都把建设秒杀系统放在了什么地位？

在头部电商平台，除了售卖我们前面讨论的爆品外，更多售卖的是普通商品，这两类商品特点鲜明，爆品具有流量激增的特点，而普通商品流量则比较均衡。

想一想，如果这两类商品不加区别，直接在电商平台上一块进行交易，会有什么问题?

没错，灾难性的后果，容易引发平台 P0 级重大事故。究其原因，主要就在于秒杀流量是突发式的，而且流量规模很难提前准确预估，如果混合在一起，势必会对普通商品的交易造成比较大的冲击。

因此，对京东、阿里而言，即使需要投入新的资源，也是需要单独搭建一套秒杀系统的，它将作为交易体系非常重要的一个核心系统。

问题三：秒杀系统对于我们意味着什么？为什么要学习秒杀系统？

秒杀系统是互联网 IT 技术人员绕不开的一个话题，大到京东、阿里这样的头部电商，小到新兴的社区团购公司，都需要通过秒杀促销活动进行拉新留存，或持续引流保持热度。因此对于互联网 IT 技术人员的同学来说，设计和开发秒杀系统就是一门必修课。

一方面，这门课程里介绍的一些高可用、高性能、高并发的设计思路遵循普适的原则，在设计其他系统时你可以举一反三；另一方面，大部分的面试场景都会考核秒杀系统的设计能力。

接下来我们就来看看头部电商的秒杀系统设计和我们的商城系统中秒杀系统的设计和实现。

秒杀业务初步分析

每年的 618、双 11 都是电商平台的专门促销日，各种营销活动、营销方式层出不穷，而秒杀就是其中最重要的手段之一。飞天茅台、华为手机、高端显卡

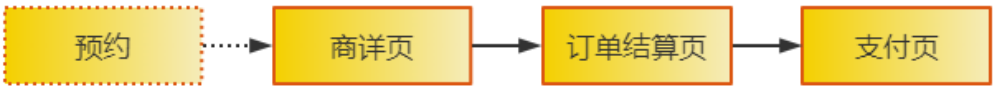
等热门商品的抢购活动，即使你没有抢过，也或许听过，这就是秒杀带来的影响力。

目的就是用具有价格优势的稀缺商品，来增加电商平台的关注度，带来空前的流量，进而可以为平台的拉新带来新助力，如果再辅以其他营销手段，比如抢购资格限制 VIP 等，那么这又是一笔可观的创收。

所以在当下这个流量为王的网络时代，能够提供秒杀的营销手段，就显得异常重要，这也是我们为什么需要做秒杀系统。

当然，实现一个秒杀系统也并不是那么容易的事，要考虑的点有很多。比如，我们首先要知道秒杀活动的业务特点，其次是要清楚秒杀系统的请求链路，这样才能根据其特点，针对请求链路中可能存在的瓶颈点做优化与设计。

通常情况下，平台商家会拿出稀缺商品，事先在秒杀的运营系统中设置好活动的开始、结束时间，以及投入的库存(这几个是秒杀主要元素)。在活动开始之后，用户可以通过活动抢购入口（一个商品详情页，或是一个广告链接），进入到活动的结算页，然后点击下单，完成商品的抢购操作，整个过程如下:



这种方式通用性很强，可以适配大部分的平台。当然如果想对流量有个预期管理，方便做备战工作，那么你可以加上预约功能，即在活动开始前，先开放一段时间的预约，让用户先去进行预约，然后才能获得参加抢购活动的资格。

如果面对的业务场景复杂些，你还可以联合风控，在参加活动时校验用户资质，踢掉黄牛以及有过不良行为的人，尽量将资源给到优质用户。

那么如果业务再复杂些呢？可以搭配限购开展活动，控制个人维度下一段时间内的购买数，让抢购触达更多的人。

以上列举的各种使用场景，可以根据自己的实际情况灵活变通，或者开拓思维创造属于自己独特的秒杀玩法。

但是在实现秒杀系统中会遇到什么样的问题和挑战呢？

秒杀系统的挑战

巨大的瞬时流量

秒杀活动的特点，就是将用户全部集中到同一个时刻，然后一起开抢某个热门商品，而热门商品的库存往往又非常少，所以持续的时间也比较短，快的话可能一两秒内就结束了。

这种场景下，高并发产生的巨大瞬时流量，首先会击垮你服务的“大门”，当“大门”被击垮后，外面的进不来，里面的出不去，进而造成了整个服务的瘫痪；紧接着如果进来的流量如果不加以管控，任凭其横冲直撞，也会对依赖的基础设施服务造成毁灭性打击；即使系统没有被摧毁，在机器资源的高负载下，整个请求链路的响应时间也会跟着拉长，这样就会大大降低用户的抢购体验，紧接

着就会是蜂拥而来的客诉。本想通过秒杀活动带来正面影响，但结果可能恰恰相反。

热点数据问题

高并发下一个无法避开的问题，就是热点数据问题。

特别是对于秒杀活动，大家抢购的都是同一个商品，所以这个商品直接就被推到了热点的位置，这对存储系统是很大的考验。像商品库存的控制，就会有这个问题。

刷子流量

一般我们提供的秒杀对外服务，都是 HTTP 的服务。不管你是用 H5 实现的页面，还是通过安卓或是 iOS 实现的原生页面，特别是 H5，都可以直接通过浏览器或是抓包工具拿到请求数据，这样刷子便可以自己通过程序实现接口的直接调用，并可以设置请求的频率。

这样高频次的请求，会挤占正常用户的抢购通道，同时，刷子也获得了更高的秒杀成功率。这不仅破坏了公平的抢购环境，也给系统服务带来了巨大的额外负担。

其实总结来说，瞬时的大流量就是最大的挑战，当业务系统流量成几何增长时，有些业务接口加机器便可以支持。但考虑到成本与收益，在有限的资源下，如何通过合理的系统设计来达到预期的业务目标，就显得格外重要了。

秒杀系统设计

清楚了秒杀系统所面临的挑战，接下来我们就可以考虑如何应对了。在设计系统之前，我们要先来看看一次 HTTP 请求所经过的链路路径:

这是一个比较宏观的图谱，如果我们提供的是一个 HTTP 服务，那么每个客户端请求进来都要经过这些链路，而每个链路节点的作用又是什么呢?我们逐一看下。

DNS：负责域名解析，会将你的域名请求指定一个实际的 IP 来处理，并且一般客户端浏览器会缓存这个 IP 一段时间，当下次再请求时就直接用这个 IP 来建立连接，当然如果指定的 IP 挂了，DNS 并不会自动剔除，下次依然会使用它。

Nginx:也就是上面的被 DNS 指定来处理请求的 IP，一般都会被用来当做反向代理和负载均衡器使用，因为它具有良好的吞吐性能，所以一般也可以用来做静态资源服务器。当 Nginx 接收到客户端请求后，根据负载均衡算法（默认是轮询）将请求分发给下游的 Web 服务。

Web 服务:这个就是我们都比较熟知的领域了，一般我们写业务接口的地方就是这了，还有我们的 H5 页面，也都可以放到这里，这里是我们做业务聚合的地方，提供页面需要的数据以及元素。

RPC 服务:一般提供支撑业务的基础服务，服务功能相对单一，可灵活、快速部署，复用性高。RPC 服务一般都是公司内部服务，仅供内部服务间调用，不对外开放，安全性高。

在了解了一次请求所经过的链路节点后，接下来我们再看下，在用户的一次抢购过程中，每次和系统的交互都要做什么事情。

支付部分，对于一般平台来说，都是通用板块，而显示商详页部分（头部机构，可能这个部分也属于通用板块）和从“点击抢购”开始到“下单成功待支付”，这一段是属于秒杀系统的业务范畴，在这里我们梳理下，有哪几件事情是和秒杀相关的。

1.秒杀的活动数据：参加秒杀活动的商品信息，主要用于商详页判断活动的倒计时、开始、结束等页面展示和抢购入口校验。

2.提供结算页：如果把秒杀做成一个单独业务模块，可跨平台(安卓、PC、iOS)嵌入，那么就需要提供一整套服务，包括 H5 页面，主要用于展示商品的抢购信息，包括商品名称、价格、抢购数量、地址、支付方式、虚拟资产等等。

3.提供结算页页面渲染所需数据：包括用户维度的地址、虚拟资产等数据，活动维度的名称、价格等数据。

4.提供下单：用户结算页下单，提供订单生成或是将下单数据透传给下游。

以上，我们了解了 HTTP 请求所经过的链路，也总结了秒杀系统所需要提供的能力，那么接下来，我们就可以着手做秒杀系统的设计了。

对于系统的设计，有一些基本的原则，比如校验前置、分层过滤。

一般大型网站会在 DNS 层做一些和网络相关的防攻击措施，网络安全部门有统一的一些配置措施，这层无法写业务也和我们没有什么太大的关系，但是可以拦截一些攻击请求。

接下来到 Nginx 层。Nginx 不仅可以作为反向代理和负载均衡器，也可以做大流量的 Web 服务器，同时也是一款非常优秀的静态资源服务器。如果把业务校验也放到这里来，就可以实现校验前置。

接下来就到了 Web 服务了。我们在这里做业务的聚合，提供结算页页面渲染所需要的数据以及下单数据透传，同时也负责流量的筛选与控制，保证下游系统的安全。

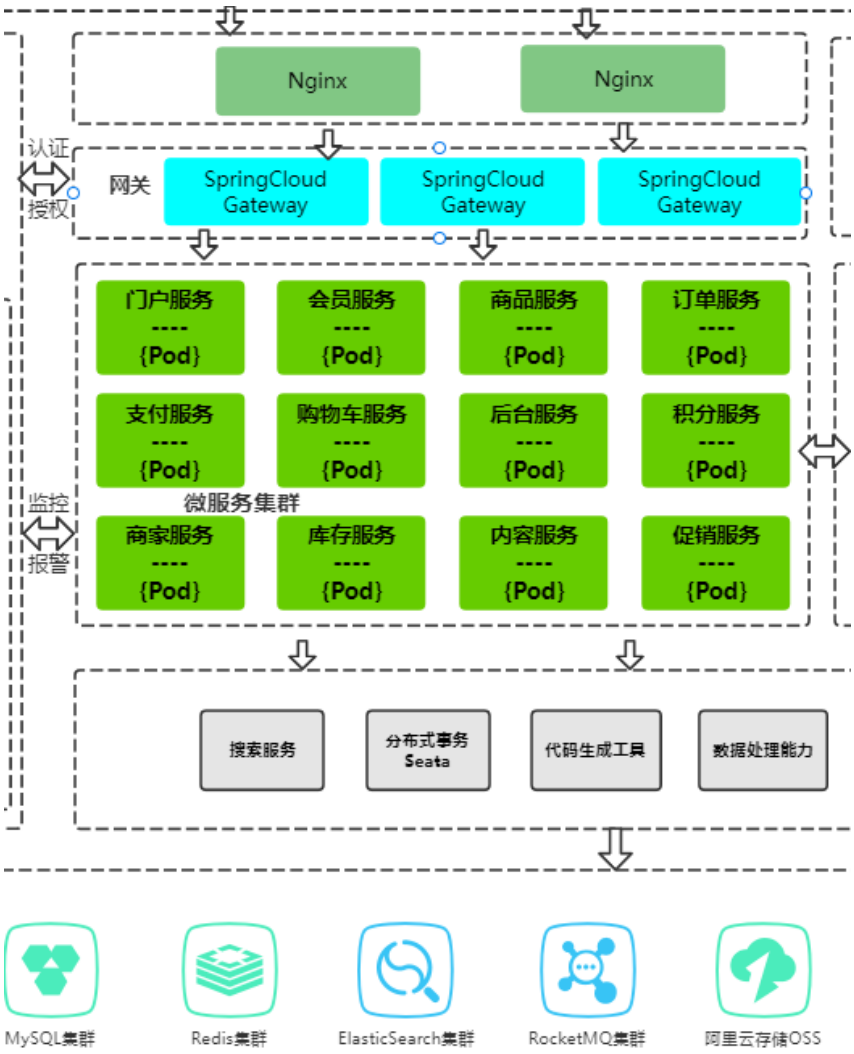
最后就是 RPC 服务。它提供基础服务，一般经过上面 3 层的严格把关，到这里的请求，量已经小很多了。

通用秒杀架构

系统的设计是个由巨入细的过程，想去设计好它，那你首先得去了解清楚它。所以这节课我们将重点分析传统架构设计的特点，接着介绍最新的秒杀系统架构，并做好技术选型和环境准备。

一般性系统架构

下面先看一个大家常用的系统功能架构图:



这种功能结构以及系统架构，是我们非常熟悉的。很多时候，Nginx 只做反向代理和负载均衡，甚至这层对大部分做业务开发的研发人员来说，都是无感知的，一般运维部门在做生产环境搭建时，都会配好。研发人员更多的是在开发 Web 服务和其他 RPC 服务/微服务，我们把页面以及页面所依赖的静态资源都放到 Web 服务中，同时 Web 服务还提供业务接口，RPC 服务提供一些支撑服务。

当然，像我们商城进行动静分离后，VUE 前端部分也会放在 Nginx 上，这就变成了页面以及页面所依赖的静态资源也在 Nginx 上，Web 服务提供业务接口，这种模式相比上面的有所改进。

对于秒杀来说瞬时流量非常大的情况，就会有很多问题，我们稍稍看几个。

页面访问

商城进行了动静分离，商详页实现在 product.vue 中。可以看到，每个商品都会去后端获得商品的详细信息并展示。

```
mounted(){
  window.scroll( x: 0, y: 0);
  this.getProductInfo();
},
methods:{
  getProductInfo(){
    let id = this.$route.params.id;

    this.axios.get( url: `/pms/productInfo/${id}`).then((res)=>{
      this.product = res;
    });
```

可以想到，这种实现的商详页在秒杀高并发的情况下，不做任何措施，会对后端服务，特别是产品服务和数据库造成非常大的访问压力，即使产品信息全部缓存，依然会消耗大量的后端资源和带宽。

Web 服务器性能问题

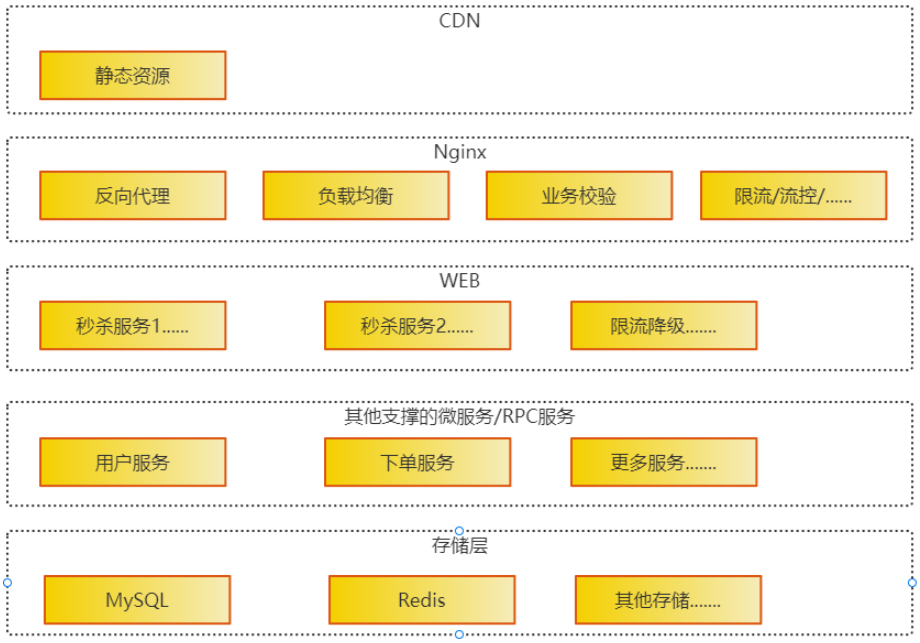
我们一般部署 Web 服务，都是使用 Tomcat 来部署的， Tomcat 在处理请求的时候，是通过线程去处理的。

这样的问题就是如果瞬时的大量请求过来，线程池中的线程不够用，Tomcat 就会瞬间新建很多线程，直至达到配置的最大线程数，如果线程数设置的过大，这个过程可能会直接将机器的 CPU 打满，导致机器死掉。即使没有挂掉，在高负载下，当设置的等待队列也满了之后，后面的请求都会被拒绝连接，直到有空出的资源去处理新请求。这时候你可能会想，我加机器分摊流量不就行了？可以是可以，但由此增加的活动成本有可能超出预算。

除此之外，还会伴有类似读写热点、库存超卖等等问题，这些我们会一一处理。

常见的秒杀系统架构

结合秒杀各链路层级，常见的大厂秒杀功能结构与系统架构图如下:



看起来似乎和一般的系统架构没什么区别，但是仔细研究区别还是很大的。一般情况下原先由 Web 服务或 Nginx 服务提供的静态资源放到了 CDN (CDN 是全国都有的服务器，客户端可以根据所处位置自动就近从 CDN 上拉取静态资源，速度更快)，来大大减轻抢购瞬时秒杀域名的负担。

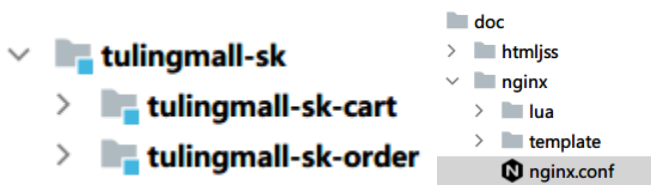
同时所做的最大改变，就是将 Nginx 的职责放大，前置用来做 Web 网关，承担部分业务逻辑校验，并且可能增加黑白名单、限流和流控的功能，这其实也是根据秒杀业务特点所做的调整。这种在 Nginx 里写业务的做法在很多大公司里都是很常见的，像京东是用来做商详、秒杀的业务网关，美团用来做负载均衡接入层，12306 用来做车票查询等等。

而这么做的目的，就是要充分利用 Nginx 的高并发、高吞吐能力，并且非常契合秒杀业务的特点，即入口流量大。但流量组成却非常的混杂，这些请求中，一部分是刷子请求，一部分是无效请求（传参等异常)，剩下的才是正常请求，一般情况下这个的比例可能是 6:1: 3，所以需要在网关层尽可能多地接收流量进来，并做精确地筛选，将真正有效的 3 成请求分发到下游，剩余的 7 成拦截在网关层。不然把这些流量都打到 Web 服务层，Web 服务再新起线程来处理刷子和无效请求，这是种资源的浪费。

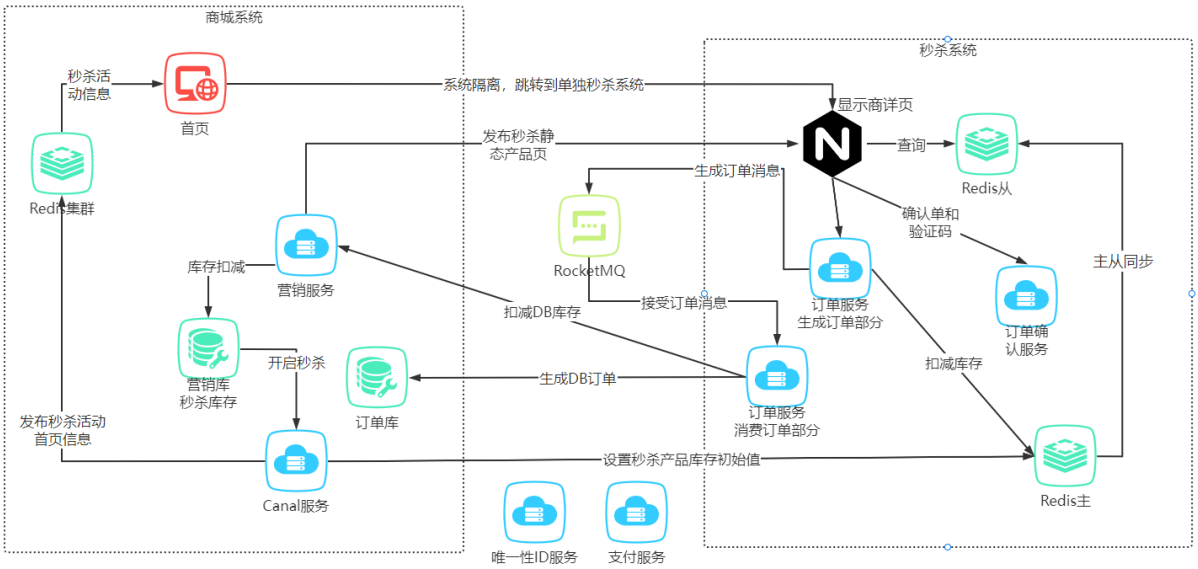
所以网关层对秒杀系统而言，至关重要，而 Nginx 刚好可以胜任此项任务。所以 Nginx 在主要的秒杀系统设计中，扮演着非常重要的角色。

商城的秒杀系统设计和实现

我们的商城秒杀系统呈现如下的代码



tulingmall-sk-cart 主要负责秒杀确认单/订单结算页处理，tulingmall-sk-order 负责秒杀订单处理，而部署完成后架构：



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。

秒杀的隔离

秒杀的隔离策略

普通商品的售卖和秒杀商品售卖最本质的区别是什么?

显而易见的是流量不同。针对普通商品，销量当然是越多越好，所以商家备货一般都会很充足，这样用户去购买的时间就会分散开，流量也会比较均衡。而秒杀商品，说白了，就是稀缺爆品，特点就是库存少，因此用户会去抢购，刷子也会热情高涨，以致瞬时流量巨大。

另外，普通商品和秒杀商品的数量级也是完全不同的。在头部电商平台，几十亿的商品都是普通商品，只有少数(百个以下)的商品具备秒杀商品的特点。

面对这样的区别，这两类商品其实很难在电商平台上一块进行交易。因为秒杀流量是突发式的，而且流量规模很难提前准确预估，如果混合在一起，势必会对普通商品的交易造成比较大的冲击。需要单独搭建秒杀系统，它天然为流量而生。

秒杀的隔离

很自然，为了不让 0.001%的爆品影响 99.999%普通商品的交易，我们很快就想到了隔离。隔离是控制危险范围的最直接的手段，正如当下新冠病毒肆虐，采取严格隔离和松散管控不同方式的不同国家，取得的效果也是完全不同的。

而面对超预期的瞬时流量，我们也要采取很多措施进行流量的隔离，防止秒杀流量串访到普通商品交易流程上，带来不可预估的灾难性后果。

业务隔离

秒杀商品的稀缺性，决定了业务不会像普通商品那样进行投放售卖。一股会有计划地进行营销策划，制订详细的方案，以达到预期的目标。

因此，从业务上看，它是和普通商品完全不一样的售卖流程，它需要一个提报过程。大部分的电商平台，会有一个专门的提报系统(提报系统的建设不是秒杀的核心部分，我们系统没有实现)，商家或者业务可以根据自己的运营计划在提报系统里进行活动提报，提供参与秒杀的商品编号、活动起止时间、库存量、限购规则、风控规则以及参与活动群体的地域分布、预计人数、会员级别等基本信息。

电商平台的提报过程和这些基本信息，对于大厂是比较重要的，有了这些信息作为输入，技术部门就能预估出大致的流量、并发数等，并结合系统当前能支撑的容量情况，评估是否需要扩容，是否需要降级或者调整限流策略等，因此业务隔离重要性也很高。

系统隔离

接下来我们看下系统隔离。前面已经介绍过商品交易流程大概会用到哪些系统，理论上讲，需要把交易链路上涉及到的系统都单独复制部署一套，隔离干净。

但这样做成本比较高，一般大点的电商平台都采用分布式微服务的部署架构，服务数量少则几十个，多则几百个，全部复制一套进行隔离不现实，我们的商城项目自然也无法做到。

所以比较常见的实践是对会被流量冲击比较大的核心系统进行物理隔离，而相对链路末端的一些系统，经过前面的削峰之后，流量比较可控了，这些系统就可以不做物理隔离。

用户的秒杀一定是首先进入商品详情页（很多电商的秒杀系统还会在商详页进行倒计时等待，时间到了点击秒杀按钮进行抢购）。因此第一个需要关注的系统就是商品详情页，我们需要申请独立的秒杀详情页域名，独立的 Nginx 负载均衡器，以及独立的详情页后端服务。

如有可能，还需要对域名进行隔离，可以申请一个独立的域名，专门用来承接秒杀流量，流量从专有域名进来之后，分配到专有的负载均衡器，再路由到专门的微服务分组，这样就做到了应用服务层面从入口到微服务的流量隔离。

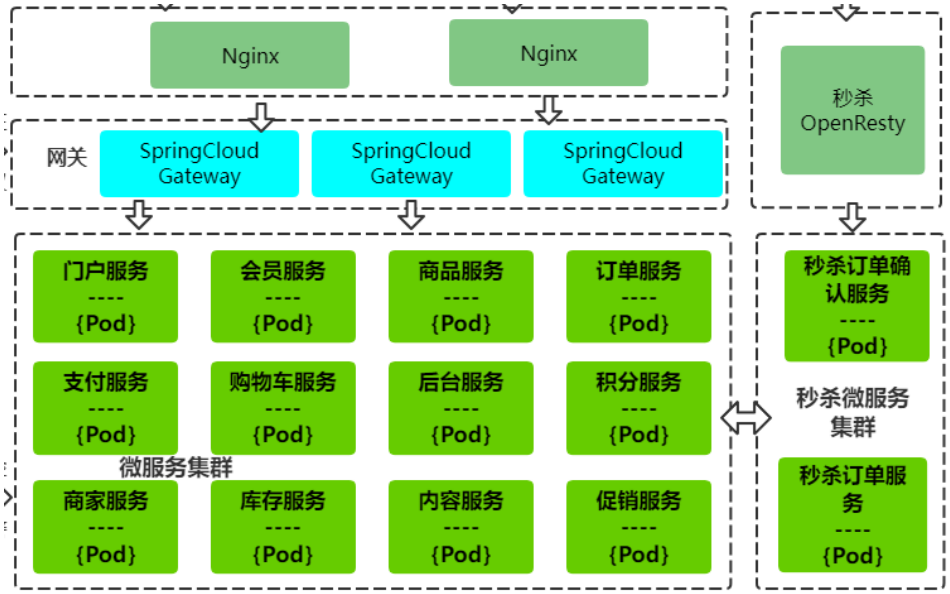
一般来说，秒杀中流量冲击比较大的核心系统就是秒杀详情页、秒杀结算页、秒杀下单库存扣减是需要我们重点关注的对象，而相对链路末端的一些系统，经过前面的削峰之后，流量比较可控了，如收银台、支付系统，物理隔离的意义就不大，反而会增加成本。

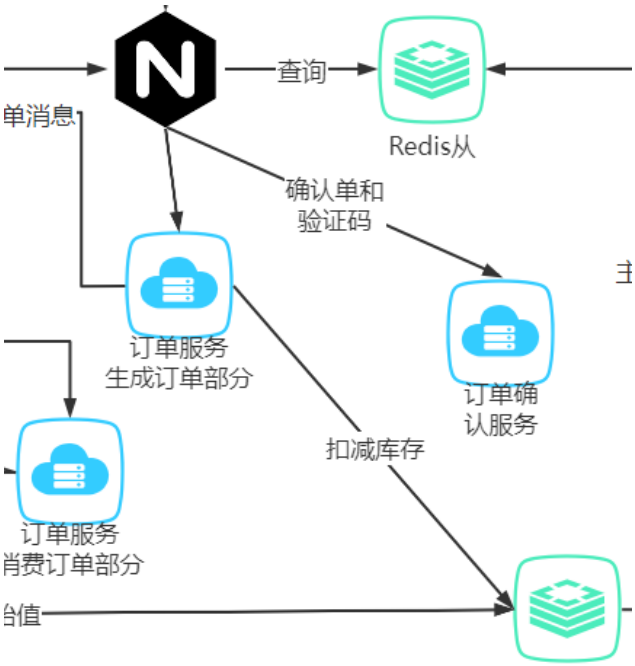
数据隔离

现在，我们已经完成了应用层的隔离。接下来，在数据层面，我们也应该进行相应的隔离，否则如果共用缓存或者共用数据库，一旦瞬时流量把它们冲垮，照样会影响无辜商品的交易。

数据层的专有部署，需要结合秒杀的场景来设计部署拓扑结构，比如 Redis 缓存，一般的场景一主一从就够了，但是在秒杀场景，需要一主多从来扛读热点数据。

实际部署





可以看到在我们的商城系统中，秒杀服务从 Nginx 服务到后面的 tulingmall-sk-cart 秒杀确认单处理服务、tulingmall-sk-order 秒杀订单处理服务都是独立部署的，而且秒杀系统拥有自己独立的 Redis 主从集群。

从首页点击秒杀栏商品首先跳转到的就是商详页，这个页面的展示由秒杀系统的第一站 Nginx 负责的，所以我们先来看看它。

OpenResty

简介

Nginx 最早被发明出来，就是来应对互联网高速发展下，出现的并发几十方、上百万的网络请求连接场景的，传统 Apache 服务器无法有效地解决这种问题，而 Nginx 却具有并发能力强、资源消耗低的特性。

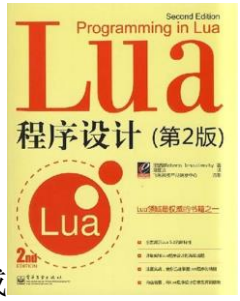
总的来说，Nginx_有 5 大优点，即模块化、事件驱动、异步、非阻塞、多进程单线程。

前面我们说过，Nginx 在主要的秒杀系统设计中，扮演着非常重要的角色，意味着 Nginx 上要承载很多的业务逻辑。Nginx 的底层模块一般都是用 C 语言写的，如果我们想在 Nginx 的基础之上写业务逻辑会很不方便，所以这个时候我们还得借助 OpenResty，它是 Nginx 的一个社区分支。OpenResty 是中国人章亦春发起，最早是雅虎中国的一个公司项目，基于 Perl 和 Haskell 实现，2007 年开始开源，后来章亦春大佬加入淘宝后进行了彻底的设计和重写。

按照官网的说法，OpenResty 是一个基于 Nginx 与 Lua 的高性能 Web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。

OpenResty 通过汇聚各种设计精良的 Nginx 模块（主要由 OpenResty 团队自主开发），从而将 Nginx 有效地变成一个强大的通用 Web 应用平台。这样，Web 开发人员和系统工程师可以使用 Lua 脚本语言调动 Nginx 支持的各种 C 以及 Lua 模块，快速构造出足以胜任 10K 乃至 1000K 以上单机并发连接的高性能 Web 应用系统。

为什么要用 Lua 语言来做 Nginx 开发呢?这就要说到 Lua 语言的特点了，Lua 的线程模型是单线程多协程的模式，而 Nginx 刚好是单进程单线程，天生的完美搭档。同时 Lua 是一种小巧的脚本语言，语法非常的简单。所以在 Redis 中也是用 Lua 作为脚本语言的。Lua 语言请自行学习，推荐两本豆瓣评分 8 分以上书籍

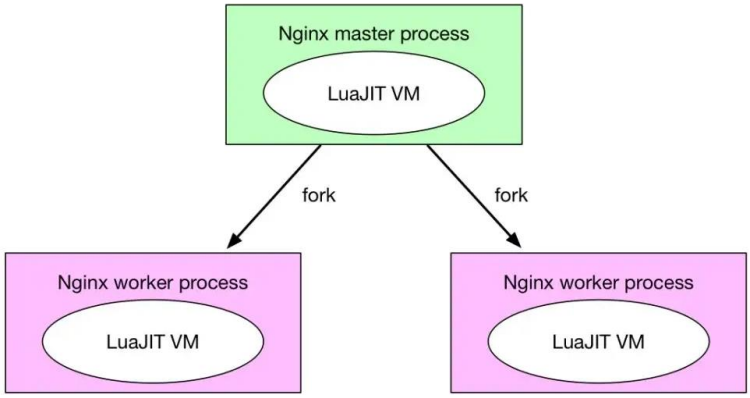


或

至于 OpenResty 的安装等知识请参考 OpenResty 中文官网：https://openresty.org/cn/

既然要使用 OpenResty ，我们还是要大概了解下 Nginx 和 OpenResty 中的一些基本原理。

原理



Nginx 服务器启动后，产生一个 Master 进程（Master Process），Master 进程执行一系列工作后产生一个或者多个 Worker 进程（Worker Processes)。其中，Master 进程用于接收来自外界的信号，并向各 Worker 进程发送信号，同时监控 Worker 进程的工作状态。当 Worker 进程退出后(异常情况下)，Master 进程也会自动重新启动新的 Worker 进程。Worker 进程则是外部请求真正的处理者。

多个 Worker 进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个 Worker 进程中处理，一个 Worker 进程不可能处理其它进程的请求。Worker 进程的个数是可以设置的，一般我们

会设置与机器 CPU 核数一致。同时，Nginx 为了更好的利用多核特性，具有 CPU 绑定选项，我们可以将某一个进程绑定在某一个核上，这样就不会因为进程的切换带来 cache 的失效（CPU affinity）。所有的进程的都是单线程（即只有一个主线程）的，进程之间通信主要是通过共享内存机制实现的。

OpenResty 本质上是将 LuaJIT 的虚拟机嵌入到 Nginx 的管理进程和工作进程中，同一个进程内的所有协程都会共享这个虚拟机，并在虚拟机中执行 Lua 代码。在性能上，OpenResty 接近或超过 Nginx 的 C 模块，而且开发效率更高。

Nginx 将 HTTP 请求的处理过程划分为多个阶段。这样可以使一个 HTTP 请求的处理过程由很多模块参与处理，每个模块只专注于一个独立而简单的功能处理，可以使性能更好、更稳定，同时拥有更好的扩展性。

1）ngx_http_post_read_phase：

接收到完整的 http 头部后处理的阶段，它位于 uri 重写之前。

2）ngx_http_server_rewrite_phase：

uri 与 location 匹配前，修改 uri 的阶段，用于重定向。

3）ngx_http_find_config_phase：

根据 uri 寻找匹配的 location 块配置项阶段，该阶段使用重写之后的 uri 来查找对应的 location，值得注意的是该阶段可能会被执行多次，因为也可能有 location 级别的重写指令。

4）ngx_http_rewrite_phase：

上一阶段找到 location 块后再修改 uri，location 级别的 uri 重写阶段，该阶段执行 location 基本的重写指令，也可能会被执行多次。

5）ngx_http_post_rewrite_phase：

防止重写 url 后导致的死循环，location 级别重写的后一阶段，用来检查上阶段是否有 uri 重写，并根据结果跳转到合适的阶段。

6）ngx_http_preaccess_phase：

下一阶段之前的准备，访问权限控制的前一阶段，该阶段在权限控制阶段之前，一般也用于访问控制，比如限制访问频率，链接数等。

7）ngx_http_access_phase：

让 http 模块判断是否允许这个请求进入 nginx 服务器，访问权限控制阶段，比如基于 ip 黑白名单的权限控制，基于用户名密码的权限控制等。

标准模块 ngx_access、第三方模块 ngx_auth_request 以及第三方模块 ngx_lua 的 access_by_lua 指令就运行在这个阶段。

8）ngx_http_post_access_phase：

访问权限控制的后一阶段，该阶段根据权限控制阶段的执行结果进行相应处理。

9）ngx_http_try_files_phase：

为访问静态文件资源而设置，try_files 指令的处理阶段，如果没有配置 try_files 指令，则该阶段被跳过。

10）ngx_http_content_phase：

处理 http 请求内容的阶段，大部分 http 模块介入这个阶段，内容生成阶段，该阶段产生响应，并发送到客户端。

Nginx 的 content 阶段是所有请求处理阶段中最为重要的一个，因为运行在这个阶段的配置指令一般都肩负着生成“内容”（content）并输出 HTTP 响应的使命。

11）ngx_http_log_phase：

log 阶段处理，比如记录访问量/统计平均响应时间。log_by_lua

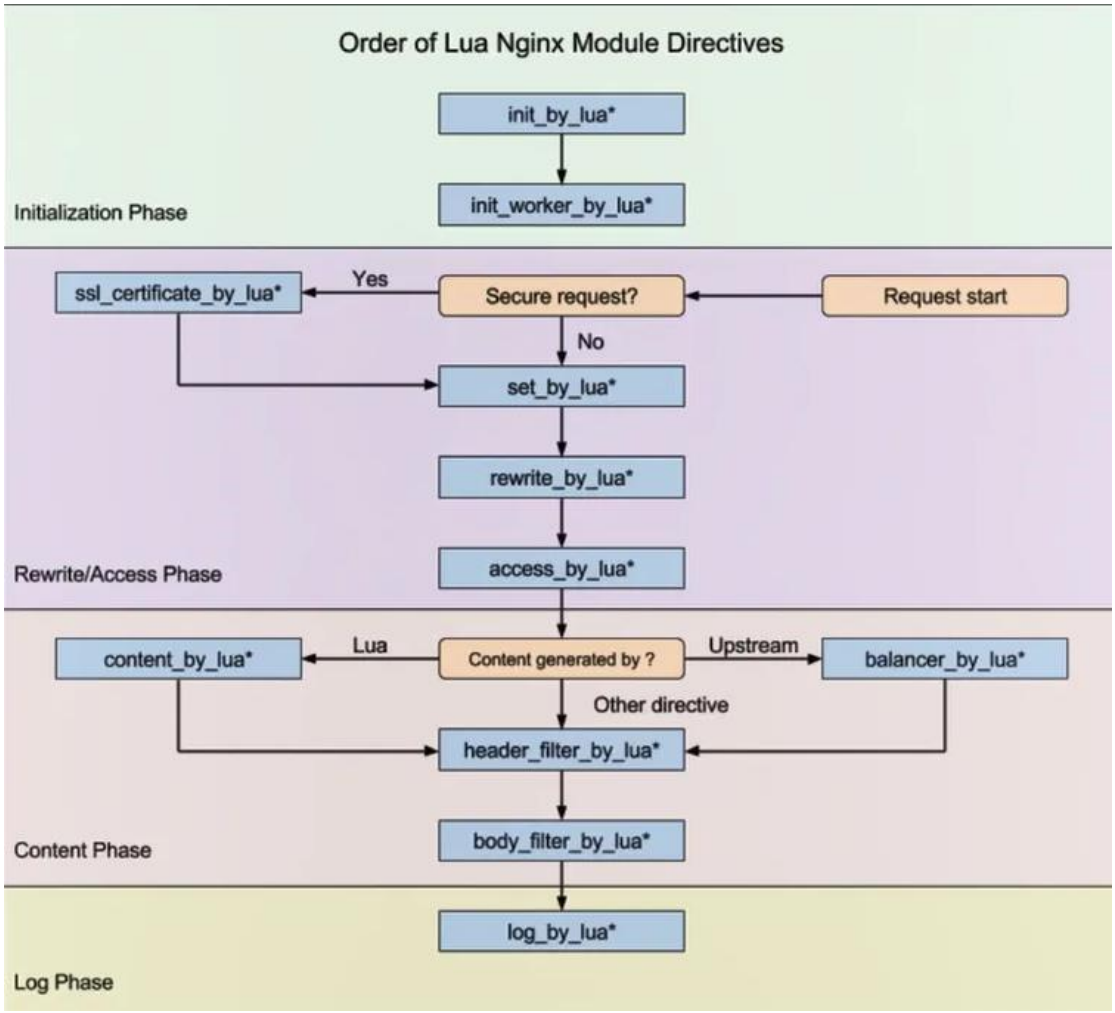
处理完请求后的日志记录阶段，该阶段记录访问日志。

以上 11 个阶段中，http 无法介入的阶段有 4 个：

3）ngx_http_find_config_phase、5）ngx_http_post_rewrite_phase、8）ngx_http_post_access_phase、9）ngx_http_try_files_phase。

OpenResty 在 HTTP 处理阶段基础上分别在 Rewrite/Access 阶段、Content 阶段、Log 阶段注册了自己的 handler，加上系统初始阶段 master 的两个阶段，共 11 个阶段为 Lua 脚本提供处理介入的能力。

Nginx 的 lua 插载点如下



其中

init_by_lua*：Master 进程加载 Nginx 配置文件时运行，一般用来注册全局变量或者预加载 Lua 模块。

init_worker_by_lua*：每个 worker 进程启动时执行，通常用于定时拉取配置/数据或者进行后端服务的健康检查。

set_by_lua*：变量初始化。

rewrite_by_lua*:可以实现复杂的转发、重定向逻辑。

access_by_lua*:IP 准入、接口权限等情况集中处理。

content_by_lua*:内容处理器，接收请求处理并输出响应。

header_filter_by_lua*:响应头部或者 cookie 处理。

body_filter_by_lua*:对响应数据进行过滤，如截断或者替换。

log_by_lua*:会话完成后，本地异步完成日志记录

商城秒杀系统中的 OpenResty

所以我们秒杀的 OpenResty 就不仅仅承担着反向代理和负载均衡的职能

```
#秒杀确认页相关负载均衡
upstream confirm {
     server 192.168.65.155:8855;
}

#秒杀订单相关负载均衡
upstream order {
     server 192.168.65.133:8844;
}
```

```
#秒杀确认页反向代理
location /skcart {
    proxy_pass http://confirm;
}

#秒杀订单反向代理
location /seckillOrder {
    proxy_pass http://order;
}
```

还承担着网关、静态模板化网页访问、静态资源访问、流量管控、防刷等一系列职能。

```
#产品静态模板化网页访问
location /product {
    default_type text/html;
    content_by_lua_file lua/product.lua;
}

#静态资源访问
location /static {
    root /usr/local/openresty;
    index index.html index.htm;
}
```

```
#秒杀产品当前库存
location /cache/stock {
    # 默认的响应类型
    default_type application/json;
    # 响应结果由lua/stock.lua文件来处理
    content_by_lua_file lua/stock.lua;
}
```

这些都需要我们使用 Lua 脚本来完成，配置一下对 Lua 脚本的支持

```
#lua 模块
lua_package_path "/usr/local/openresty/lua/?.lua;/usr/local/openresty/lualib/?.lua;;";
#c模块
lua_package_cpath "/usr/local/openresty/lualib/?.so;;";
lua_code_cache on;
```

商详页的静态化

前面说过商城进行了动静分离，商详页实现在 product.vue 中，每个商品都会去后端获得商品的详细信息并展示。在秒杀高并发的情况下，不做任何措施，会对后端服务造成非常大的访问压力。

仔细想一下，我们知道，进行秒杀的商品是确定的，和秒杀商品相关的属性，比如规格、图片地址等都是确定的，在用户进入秒杀商详页时并不需要每次都从后端服务获取，在我们的商城项目中，唯一需要每次获取的只有商品的当前秒杀库存。那么我们为什么不把秒杀的商品详情页静态化呢，这样可以充分利用 Nginx 对静态网页的高性能处理能力。

tulingmall-promotion 中的处理

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。然后活动开始之前，由秒杀系统运营后台开启秒杀。

很明显，每次秒杀的商品都是不一样的，很难每次秒杀时手工制作，还是需要程序来处理。

tulingmall-promotion 中的 SecKillController 提供了开启秒杀接口

```
@ApiOperation("开启秒杀")
@RequestMapping(value = "/openSecKill", method = RequestMethod.GET)
@ResponseBody
public CommonResult<Integer> turnOnSecKill(@RequestParam long secKillId){
    int result = homePromotionService.turnOnSecKill(secKillId, ConstantPromotion.SECKILL_OPEN);
    try {
        if(secKillStaticHtmlService.deployHtml(secKillId) == ConstantPromotion.STATIC_HTML_FAILURE){
            return CommonResult.failed("发布秒杀静态页失败！");
        }
    } catch (Exception e) {
        log.error("发布秒杀静态产品页异常：",e);
        homePromotionService.turnOnSecKill(secKillId, ConstantPromotion.SECKILL_CLOSE);
        return CommonResult.failed(e.getMessage());
    }
    return CommonResult.success(result);
}
```

SecKillStaticHtmlServiceImpl 负责初步的网页静态化和部署

```java
@Override
public int deployHtml(long secKillId) throws Exception {
    List<String> result = makeStaticHtml(secKillId);
    if(!CollectionUtils.isEmpty(result)){
        for(String host : nginxServerList){
            ChannelSftp channel = sftpUploadService.getChannel(host, userName, por
            String path = rootPath + "/";
            sftpUploadService.createDir(path,channel);
            for(String fileName : result){
                sftpUploadService.putFile(channel,new FileInputStream( name: htmlDi
            }
            channel.quit();
            channel.exit();
            log.info("服务器：{}，静态网页上传完成",host);
        }
        return ConstantPromotion.STATIC_HTML_SUCCESS;
    }else{
        return ConstantPromotion.STATIC_HTML_FAILURE;
    }
}
```

首先当然是生成静态 html，我们使用 freemarker 来处理，将通用的产品模板文件 product.ftl 变为确定的商品 Html，这个由 makeStaticHtml()方法和 toStatic()方法负责，最后生成的商品 Html，命名规则为 seckill_+秒杀活动 id + "_" + 秒杀产品 ID，如 seckill_1_3.html，并保存在本地磁盘。

```java
/*根据秒杀活动，静态化该秒杀活动的所有页面*/
1 usage   Mark
@Override
public List<String> makeStaticHtml(long secKillId) throws TemplateException, IOExcepti
    log.info("本地模板目录：{}，本地html目录：{}",templateDir,htmlDir);
    //查询秒杀商品信息
    List<FlashPromotionProduct> flashPromotionProducts =
            homePromotionService.secKillContent(secKillId,ConstantPromotion.SECKILL_OP
    List<String> result = new ArrayList<>();
    if(CollectionUtils.isEmpty(flashPromotionProducts)){
        log.warn("没有秒杀活动{[]}对应的产品信息，请检查DB中的秒杀数据",secKillId);
    }else{
        for(FlashPromotionProduct flashPromotionProduct : flashPromotionProducts){
            result.add(toStatic(flashPromotionProduct));
```

```
/*具体产品页面的静态化*/
1 usage   Mark
private String toStatic(FlashPromotionProduct flashPromotionProduct) throws IOException, Tem
    String outPath = "";
    // 第一步：创建一个Configuration对象，直接new一个对象。构造方法的参数就是freemarker对于的
    Configuration configuration = new Configuration(Configuration.getVersion());
    // 第二步：设置模板文件所在的路径。
    configuration.setDirectoryForTemplateLoading(new File(templateDir));
    // 第三步：设置模板文件使用的字符集。一般就是utf-8.
    configuration.setDefaultEncoding("utf-8");
    // 第四步：加载一个模板，创建一个模板对象。
    Template template = configuration.getTemplate(templateName);
    // 第五步：创建一个模板使用的数据集，可以是pojo也可以是map。一般是Map。
    Map dataModel = new HashMap();
    // 向数据集中添加数据
    dataModel.put("fpp", flashPromotionProduct);

    String images = flashPromotionProduct.getPic();
    if (StringUtils.isNotEmpty(images)) {
        String[] split = images.split( regex: ",");
        List<String> imageList = Arrays.asList(split);
        dataModel.put("imageList", imageList);
    }
    // 第六步：创建一个Writer对象，一般创建一FileWriter对象，指定生成的文件名。
    // 文件名命名规则  seckill_+秒杀活动id + "_" + 秒杀产品ID，如 seckill_1_3.html
    String fileName = "seckill_" + flashPromotionProduct.getFlashPromotionId() + "_" + flashProm
```

在 deployHtml()方法中将产生的商品 Html 上传到 Nginx 服务器，这里利用了 jsch 组件

```
<!-- 文件上传组件 -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
<dependency>
    <groupId>com.jcraft</groupId>
    <artifactId>jsch</artifactId>
    <version>0.1.54</version>
</dependency>
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.10.3</version>
</dependency>
```

并且由 SftpUploadService 服务利用 sftp 进行上传，注意，因为秒杀的 Nginx 服务器可能有多台，所以要循环服务器列表依次上传。

ps：这里其实有个模拟的优化，我们知道在实际秒杀中，秒杀有单独的域名，在 DNS 域名解析时可以负载均衡到不同的秒杀 Nginx 服务器。但是我们这里没有 DNS 服务，所以在生成首页秒杀概略图内容时，指向秒杀 Nginx 服务的链接使用了类似DNS 域名解析的方案，让多个秒杀商品的访问落在不同的Nginx 服务之上，表现在 HomePromotionServiceImpl 的 secKillContent()的方法之中：

```
String url = secKillServerList.get(loop % serverSize)+"/product?"+"flashPromotionId="+flashPromotionId
        +"&promotionProductId="+productId;
```

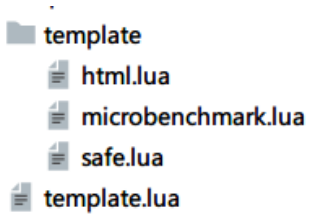
OpenResty 中的处理

此时，Nginx 服务器上已经有了本次秒杀活动商品的 Html 页面

```
[root@192-168-65-28 tpl]# ls
seckill_3_26.html  seckill_3_27.html  seckill_3_29.html  seckill_3_32.html  seckill_3_42.html
[root@192-168-65-28 tpl]# pwd
/usr/local/openresty/tpl
```

但是当用户访问商详页时，传入的链接一般是这种形式 http://192.168.65.28/product?flashPromotionId=3&promotionProductId=29&memberId=1

我们还需要把访问链接和实际的 Html 页面进行对应，而且在把页面作为结果返回给用户之前，我们可能还有一些其他的事情要做，这就需要在 Nginx 中再将已经生成的 seckill_1_3.html 作为模板文件再处理一次。于是我们引入第三方的模板文件处理 Lua 脚本，并放置到 Nginx 服务器 Lua 库目录下




```
[root@192-168-65-28 resty]# ls
aes.lua   dns       lrucache      memcached.lua  redis.lua   sha256.lua  sha.lua     string.lua    upload.lua
core      limit     lrucache.lua  mysql.lua      sha1.lua    sha384.lua  shell.lua   template      upstream
core.lua  lock.lua  md5.lua       random.lua     sha224.lua  sha512.lua  signal.lua  template.lua  websocket
[root@192-168-65-28 resty]# pwd
/usr/local/openresty/lualib/resty
[root@192-168-65-28 resty]#
```

编写了我们自己的 product.lua

```
-- 导入lua-resty-template函数库
local template = require('resty.template')
local flashPromotionId = ngx.var.arg_flashPromotionId
ngx.log(ngx.ERR, "秒杀活动ID: ", flashPromotionId)
local promotionProductId = ngx.var.arg_promotionProductId
ngx.log(ngx.ERR, "秒杀产品ID: ", promotionProductId)
local templateName = "seckill_"..flashPromotionId.."_"..promotionProductId..".html"
local context = {
    memberId = ngx.var.arg_memberId,
    productId = promotionProductId,
    flashPromotionId = flashPromotionId
}
ngx.log(ngx.ERR, "渲染页面输出，获得当前用戶ID: ", context.memberId)
template.render(templateName, context)
```

当然，当用户访问秒杀的商详页时，还需要告诉 Nginx 返回给用户的网页内容需要由 product.lua 进行处理

```
#产品静态模板化网页访问
location /product {
    default_type text/html;
    content_by_lua_file lua/product.lua;
}
```

商详页的库存获取

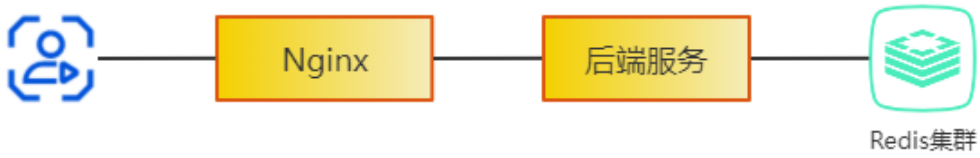
用户访问秒杀的商详页，我们用静态网页展示给了用户，但是有些数据还是需要动态获取的，比如秒杀商品的库存。

虽然库存在生成静态网页时已经初始化为商品的活动预设库存数

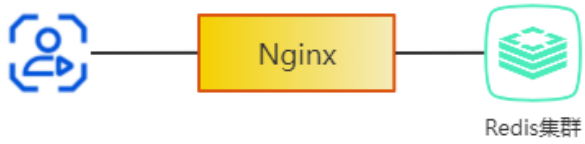
```
<span>
  <em>库存数量</em>
</span>
<span>
  <input id="flashPromotionCount" type="text" readonly value="${fpp.flashPromotionCount}">
```

但是随着活的的进行，这个库存数肯定会发生变化，我们当然希望尽可能的及时反应库存的变化。

从前面的秒杀架构图我们已经知道秒杀系统的 Redis 主从集群会有秒杀商品库存的信息，很明显，按照我们一般的做法，访问路径应该是这样的

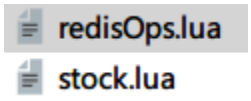


但是仔细想想，有必要吗？后端服务在这里只是起了一个转发的作用，为什么我们不直接让 Nginx 访问 Redis 来获得商品的当前库存呢？



这样的话，既降低了后端服务的压力，又提升了秒杀系统的性能，所以我们在 OpenResty 中查询库存的时候，直接访问了 Redis。

```
#秒杀产品当前库存
location /cache/stock {
    # 默认的响应类型
    default_type application/json;
    # 响应结果由lua/stock.lua文件来处理
    content_by_lua_file lua/stock.lua;
}
```



其中 redisOps.lua 是 OpenResty 访问 Redis 的方法的封装（RedisExtOps.lua 是更全面的封装，支持 pipeline、sub/pub 等操作，OpenResty 甚至还支持直接访问 MySQL，对应的文件为 MySQLOps.lua，但是 RedisExtOps.lua 和 MySQLOps.lua 都未经测试），stock.lua 则在 redisOps.lua 的基础之上加入了业务部分，访问了 Redis 集群中商品的库存值

```
-- 导入redisOps函数库
local redisOps = require('redisOps')
local read_redis = redisOps.read_redis
```

```
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: ", k
        -- 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
end

-- 获取请求参数中的productId，也可以使用ngx.req.get_uri_args["productId"
local product_id = ngx.var.arg_productId

-- 查询库存信息
local stock = read_data("miaosha:stock:cache:"..product_id, 3600)
```

更进一步，Nginx 要通过网络访问 Redis，能不能连这个网络访问都避免呢？既然秒杀中我们会搭建 Redis 主从集群，为什么我们不让 Redis 的从库和 Nginx 在部署在同一台服务器呢？

```
[root@192-168-65-28 local]# ps -ef|grep -E "nginx|redis"
nobody    2938  5657  0 9月29 ?        00:00:00 nginx: worker process
nobody    2939  5657  0 9月29 ?        00:00:00 nginx: worker process
nobody    2940  5657  0 9月29 ?        00:00:00 nginx: worker process
nobody    2941  5657  0 9月29 ?        00:00:00 nginx: worker process
nobody    2942  5657  0 9月29 ?        00:00:00 nginx: worker process
nobody    2943  5657  0 9月29 ?        00:00:00 nginx: worker process
nobody    2944  5657  0 9月29 ?        00:00:00 nginx: worker process
root      5657     1  0 9月14 ?        00:00:00 nginx: master process ./nginx/sbin/nginx -c conf/ng
root     15322 15137  0 14:51 pts/0    00:00:00 grep --color=auto -E nginx|redis
root     17809     1  0 9月17 ?        03:41:11 ./src/redis-server *:6379
```

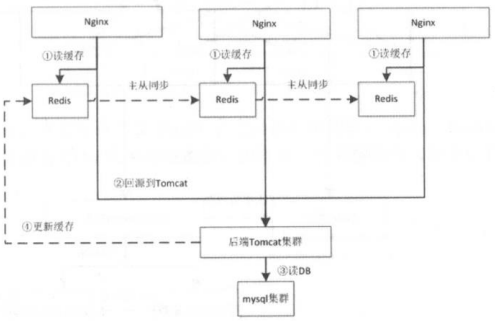
```
# In some cases redis will emit warning
# that the system is in bad state, it i
# by setting the following config which
# to suppress
#
# ignore-warnings ARM64-COW-BUG
slaveof 192.168.65.190 6379
slave-read-only yes
```

这样的话 Nginx 访问 Redis 时，最多经过操作系统网络协议栈的 IP 层即可完成数据的访问，避免了数据包在网络上的实际传输。事实上，京东内部就使用了这种设计，《亿级流量网站架构核心技术 张开涛著》就有相关的介绍，在第 351 页和 385 页

2．读取本地 Redis 数据架构

如下图所示，可以看到 Nginx 应用和 Redis 集群部署在同一台机器上，这样的好处是可以消除跨机器、跨交换机或跨机柜，甚至跨机房调用。如果本地 Redis 集群不命中，则还是回源到 Tomcat 集群进行取数据。此种方式可能受限于 TCP 连接数，可以考虑使

20.2.2 OpenResty+Local Redis+MySQL 集群架构



如上图所示，OpenResty 首先通过 Lua 读取本机 Redis 缓存，如果不命中，则回源到后端 Tomcat 集群。后端 Tomcat 集群再读取 MySQL 数据库。Redis 都是安装到和

甚至于我们还可以直接使用 Unix Domain Socket 来避免真实的网络通讯实现下占用网络连接、并且需要经过网络协议栈，需要打包拆包、计算校验和、维护序号和应答等 TCPIP 协议固有要求，进一步提高访问效率。当然，要实现 Unix Domain Socket ，对 Lua 语言要求较高，我们暂时不考虑这种方式。

所以在我们的 stock.lua 中可以看到我们访问的 Redis 的地址为 127.0.0.1，当然我们的实现并不完全，因为还没有考虑如果本地 Redis 宕机的情况，这个时候需要我们回源到微服务中查询主 Redis 或者数据库，但是我们只是简单视同商品被秒光的情况，告诉用户秒杀已结束，处理略显粗糙。

```
$.get("cache/stock?productId="+$("#productId").val(),function (data) {
  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;
  }
})
```

但是在 redisOps.lua 中我们已经预留了访问后端服务的接口

```
-- 封装函数，发送http请求，并解析响应
local function read_http(path, params)
    local resp = ngx.location.capture(path,{
        method = ngx.HTTP_GET,
        args = params,
    })
    if not resp then
        -- 记录错误信息，返回404
        ngx.log(ngx.ERR, "http查询失败，path: ", path , ", args: ", args)
        ngx.exit(404)
    end
    return resp.body
end
```

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？

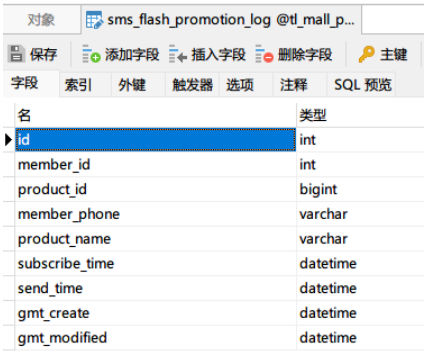
通过对秒杀流量的隔离，我们已经能够把巨大瞬时流量的影响范围控制在隔离的秒杀环境里了。接下来，我们开始考虑隔离环境的高可用问题，通俗点说，普通商品交易流程保住了，现在就要看怎么把秒杀系统搞稳定，来应对流量冲击，让秒杀系统也不出问题。方法很多，有流量控制、削峰、限流、缓存热点处理、扩容、熔断等一系列措施。

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



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

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。

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

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

```
@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：

```
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，另一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。

tulingmall-sk-order 秒杀订单处理的 SecKillOrderServiceImpl 的生成秒杀订单方法 generateSecKillOrder()中就使用了 RocketMQ 异步下单，以避免对订单数据库造成冲击

```
try {
    boolean sendStatus = orderMessageSender.sendCreateOrderMsg(orderMessage);
    if(sendStatus){
        /*打上排队的标记*/
        redisStockUtil.set( key: RedisKeyPrefixConst.MIAOSHA_ASYNC_WAITING_PRE
                ,Integer.toString( i: 1), timeout: 60, TimeUnit.SECONDS);
        result.put("orderStatus",OrderConstant.ORDER_SECKILL_ORDER_TYPE_ASYN)
    }else{
        failSendMessage(productId,result);
        return CommonResult.failed(result, message: "下单失败，请稍后再试");
    }
} catch (Exception e) {
    log.error("消息发送失败:error msg:{}",e.getMessage(),e.getCause());
    failSendMessage(productId,result);
    return CommonResult.failed(result, message: "下单失败，请稍后再试");
}
```

但是异步下单后就要注意了，因为是异步生成订单，当用户支付时还不能保证数据库中这张订单已经实际生成，所以需要前端定期去查询结果反馈给用户。所以在 secKillPay.html 里可以看到这样的实现：

```
  $.ajax({
    type: "post",
    headers:{"memberId": lMemberId},//设置请求头
    url:"/seckillOrder/checkOrder?" + "orderId="+orderId,
    success:function(commonresult){
      if(commonresult.code==200){
        console.log("秒杀订单已生成！");
        $("payImage").attr("hidden",false);
        $("showProgress").attr("hidden",true);
      }else if(commonresult.code==202){
        console.log("秒杀订单还未生成，继续检查！");
        checkCount++;
      }else{
        alert("检查秒杀订单生成情况失败！");
      }
    },
    error:function(data){
      alert("检查秒杀订单生成情况异常！");
    }
  });
}

timer = setInterval(checkOrder,5000);
```

而在秒杀系统拉取消息队列进行处理的时候，也有个小技巧，那就是当前面的请求已经把库存消耗光之后，在缓存里设置占位符，让后续的请求快速失败，从而最快地进行响应。

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

在前面的《订单系统的设计与海量数据处理实战》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

```
-- 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

```
/*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();
    }
}
```

```
<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.getCho
            }
        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—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。

本文档分享地址：

http://note.youdao.com/noteshare?id=4e763fe2734946d6dc234e38511f041a&sub=B8A621BED2334F4387DAD2866F134A1B