一、秒杀系统架构设计都有哪些关键点？

说实话，作为一名程序员，我的技术能力也在公司业务的快速增长过程中得到了历练，并积累了一些大流量高并发网站架构设计和优化的经验，尤其是针对“秒杀”这个场景。因为我确信，那个时候我们肯定是对系统做了足够多的极致优化，才能扛住当时洪峰般的流量请求。

记得早期的时候，淘宝商品详情系统的 PV 还差不多是 1 亿的样子，但是到 2016 年差不多已经升至 50 亿了。尤其是 2012 年到 2014 年那个时间段，“秒杀”活动特别流行，用户的参与热情一浪高过一浪，系统要面对的流量也是成倍增长。

而每一次的秒杀活动对技术团队来说都是一次考验。现在想起来，那个时候我们整个团队，无所畏惧，逐步迭代创新，然后解决一个个难题的过程，也是极具挑战性和成就感的事情。

记得有一年，为了应对“双十一”，我们整个商品详情团队对系统做了很多优化，我们自认为已经是整个公司最牛的系统了，性能也已经是“业界之巅”。

但是那年“双十一”的晚上，我们的系统还是遇到了瓶颈。当时老大就跑过来盯着我们，问我们什么时候能够恢复，我们整个团队都承担着巨大的心理压力。

事后我们复盘宕机的原因，发现当时的秒杀流量远远超过了我们的预想，我们根本没想到大家的参与热情能有那么高。于是我们按照这个增长率去预估下一年的流量和服务器，粗算下来，我记得差不多要增加 2000 台服务器，简直不可思议。

怎么可能真正增加这么多机器，所以这也就倒逼我们必须找出一些特殊的手段来优化系统。后面，经过一段时间的调研和分析，我们想到了把整个系统进行动静分离改造的解决方案。

秒杀系统也差不多那个时候才从商品详情系统独立出来成为一个独立产品的。因为我见证了秒杀系统的建设过程，所以也有颇多感慨。秒杀系统的迭代又是一个升级打怪的过程，我们也都是遇到问题解决问题，逐一优化。

那么，如何才能更好地理解秒杀系统呢？我觉得作为一个程序员，你首先需要从高维度出发，从整体上思考问题。在我看来，**秒杀其实主要解决两个问题，一个是并发读，一个是并发写**。并发读的核心优化理念是尽量减少用户到服务端来“读”数据，或者让他们读更少的数据；并发写的处理原则也一样，它要求我们

在数据库层面独立出来一个库，做特殊的处理。另外，我们还要针对秒杀系统做一些保护，针对意料之外的情况设计兜底方案，以防止最坏的情况发生。

而从一个架构师的角度来看，要想打造并维护一个超大流量并发读写、高性能、高可用的系统，在整个用户请求路径上从浏览器到服务端我们要遵循几个原则，就是要保证用户请求的数据尽量少、请求数尽量少、路径尽量短、依赖尽量少，并且不要有单点。这些关键点我会在后面的文章里重点讲解。

其实，秒杀的整体架构可以概括为“稳、准、快”几个关键字。

所谓“稳”，就是整个系统架构要满足高可用，流量符合预期时肯定要稳定，就是超出预期时也同样不能掉链子，你要保证秒杀活动顺利完成，即秒杀商品顺利地卖出去，这个是最基本的前提。

然后就是“准”，就是秒杀 10 台 iPhone，那就只能成交 10 台，多一台少一台都不行。一旦库存不对，那平台就要承担损失，所以“准”就是要求保证数据的一致性。

最后再看“快”，“快”其实很好理解，它就是说系统的性能要足够高，否则你怎么支撑这么大的流量呢？不光是服务端要做极致的性能优化，而且在整个请求链路上都要做协同的优化，每个地方快一点，整个系统就完美了。

所以从技术角度上看“稳、准、快”，就对应了我们架构上的高可用、一致性和高性能的要求，我们的专栏也将主要围绕这几个方面来展开，具体如下。

高性能。 秒杀涉及大量的并发读和并发写，因此支持高并发访问这点非常关键。本专栏将从设计数据的动静分离方案、热点的发现与隔离、请求的削峰与分层过滤、服务端的极致优化这 4 个方面重点介绍。

一致性。 秒杀中商品减库存的实现方式同样关键。可想而知，有限数量的商品在同一时刻被很多倍的请求同时来减库存，减库存又分为“拍下减库存”“付款减库存”以及预扣等几种，在大并发更新的过程中都要保证数据的准确性，其难度可想而知。因此，我将用一篇文章来专门讲解如何设计秒杀减库存方案。

高可用。 虽然我介绍了很多极致的优化思路，但现实中总难免出现一些我们考虑不到的情况，所以要保证系统的高可用和正确性，我们还要设计一个 PlanB 来兜底，以便在最坏情况发生时仍然能够从容应对。专栏的最后，我将带你思考可以从哪些环节来设计兜底方案。

最后，很幸运能在极客时间遇到你，希望这堂课能让你彻底理解大并发、高性能、高可用秒杀系统的设计之道，并能够在思考解决类似问题时有更准确的思考和判断。

二、设计秒杀系统时应该注意的 5 个架构原则

说起秒杀，我想你肯定不陌生，这两年，从双十一购物到春节抢红包，再到 12306 抢火车票，“秒杀”的场景处处可见。简单来说，秒杀就是在同一个时刻有大量的请求争抢购买同一个商品并完成交易的过程，用技术的行话来说就是大量的并发读和并发写。

不管是哪一门语言，并发都是程序员们最为头疼的部分。同样，对于一个软件而言也是这样，你可以很快增删改查做出一个秒杀系统，但是要让它支持高并发访问就没那么容易了。比如说，如何让系统面对百万级的请求流量不出故障？如何保证高并发情况下数据的一致性写？完全靠堆服务器来解决吗？这显然不是最好的解决方案。

在我看来，**秒杀系统本质上就是一个满足大并发、高性能和高可用的分布式系统**。今天，我们就来聊聊，如何在满足一个良好架构的分布式系统基础上，针对秒杀这种业务做到极致的性能改进。

2.1 架构原则：“4 要 1 不要”

如果你是一个架构师，你首先要勾勒出一个轮廓，想一想如何构建一个超大流量并发读写、高性能，以及高可用的系统，这其中有哪些要素需要考虑。我把这些要素总结为“4 要 1 不要”。

2.1.1. 数据要尽量少

所谓“数据要尽量少”，首先是指用户请求的数据能少就少。请求的数据包括上传给系统的数据和系统返回给用户的数据（通常就是网页）。

为啥“数据要尽量少”呢？因为首先这些数据在网络上传输需要时间，其次不管是请求数据还是返回数据都需要服务器做处理，而服务器在写网络时通常都要做压缩和字符编码，这些都非常消耗 CPU，所以减少传输的数据量可以显著减少 CPU 的使用。例如，我们可以简化秒杀页面的大小，去掉不必要的页面装修效果，等等。

其次，“数据要尽量少”还要求系统依赖的数据能少就少，包括系统完成某些业务逻辑需要读取和保存的数据，这些数据一般是和后台服务以及数据库打交道的。调用其他服务会涉及数据的序列化和反序列化，而这也是 CPU 的一大杀手，

同样也会增加延时。而且，数据库本身也容易成为一个瓶颈，所以和数据库打交道越少越好，数据越简单、越小则越好。

2.1.2. 请求数要尽量少

用户请求的页面返回后，浏览器渲染这个页面还要包含其他的额外请求，比如说，这个页面依赖的 CSS/JavaScript、图片，以及 Ajax 请求等等都定义为“额外请求”，这些额外请求应该尽量少。因为浏览器每发出一个请求都多少会有一些消耗，例如建立连接要做三次握手，有的时候有页面依赖或者连接数限制，一些请求（例如 JavaScript）还需要串行加载等。另外，如果不同请求的域名不一样的话，还涉及这些域名的 DNS 解析，可能会耗时更久。所以你要记住的是，减少请求数可以显著减少以上这些因素导致的资源消耗。

例如，减少请求数最常用的一个实践就是合并 CSS 和 JavaScript 文件，把多个 JavaScript 文件合并成一个文件，在 URL 中用逗号隔开（https://g.xxx.com/tm/xx-b/4.0.94/mods/??module-preview/index.xtpl.js,module-jhs/index.xtpl.js,module-focus/index.xtpl.js）。这种方式在服务端仍然是单个文件各自存放，只是服务端会有一个组件解析这个 URL，然后动态把这些文件合并起来一起返回。

2.1.3. 路径要尽量短

所谓“路径”，就是用户发出请求到返回数据这个过程中，需求经过的中间的节点数。

通常，这些节点可以表示为一个系统或者一个新的 Socket 连接（比如代理服务器只是创建一个新的 Socket 连接来转发请求）。每经过一个节点，一般都会产生一个新的 Socket 连接。

然而，每增加一个连接都会增加新的不确定性。从概率统计上来说，假如一次请求经过 5 个节点，每个节点的可用性是 99.9% 的话，那么整个请求的可用性是：99.9% 的 5 次方，约等于 99.5%。

所以缩短请求路径不仅可以增加可用性，同样可以有效提升性能（减少中间节点可以减少数据的序列化与反序列化），并减少延时（可以减少网络传输耗时）。

要缩短访问路径有一种办法，就是多个相互强依赖的应用合并部署在一起，把远程过程调用（RPC）变成 JVM 内部之间的方法调用。在《大型网站技术架构演进与性能优化》一书中，我也有一章介绍了这种技术的详细实现。

2.1.4. 依赖要尽量少

所谓依赖，指的是要完成一次用户请求必须依赖的系统或者服务，这里的依赖指的是强依赖。

举个例子，比如说你要展示秒杀页面，而这个页面必须强依赖商品信息、用户信息，还有其他如优惠券、成交列表等这些对秒杀不是非要不可的信息（弱依赖），这些弱依赖在紧急情况下就可以去掉。

要减少依赖，我们可以给系统进行分级，比如 0 级系统、1 级系统、2 级系统、3 级系统，0 级系统如果是最重要的系统，那么 0 级系统强依赖的系统也同样是最重要的系统，以此类推。

注意，0 级系统要尽量减少对 1 级系统的强依赖，防止重要的系统被不重要的系统拖垮。例如支付系统是 0 级系统，而优惠券是 1 级系统的话，在极端情况下可以把优惠券给降级，防止支付系统被优惠券这个 1 级系统给拖垮。

2.1.5. 不要有单点

系统中的单点可以说是系统架构上的一个大忌，因为单点意味着没有备份，风险不可控，我们设计分布式系统最重要的原则就是“消除单点”。

那如何避免单点呢？我认为关键点是避免将服务的状态和机器绑定，即把服务无状态化，这样服务就可以在机器中随意移动。

如何那把服务的状态和机器解耦呢？这里也有很多实现方式。例如把和机器相关的配置动态化，这些参数可以通过配置中心来动态推送，在服务启动时动态拉取下来，我们在这些配置中心设置一些规则来方便地改变这些映射关系。

应用无状态化是有效避免单点的一种方式，但是像存储服务本身很难无状态化，因为数据要存储在磁盘上，本身就要和机器绑定，那么这种场景一般要通过冗余多个备份的方式来解决单点问题。

前面介绍了这些设计上的一些原则，但是你有没有发现，我一直说的是“尽量”而不是“绝对”？

我想你肯定会问是不是请求最少就一定最好，我的答案是“不一定”。我们曾经把有些 CSS 内联进页面里，这样做可以减少依赖一个 CSS 的请求从而加快首页的渲染，但是同样也增大了页面的大小，又不符合“数据要尽量少”的原则，这

种情况下我们为了提升首屏的渲染速度，只把首屏的 HTML 依赖的 CSS 内联进来，其他 CSS 仍然放到文件中作为依赖加载，尽量实现首屏的打开速度与整个页面加载性能的平衡。

所以说，**架构是一种平衡的艺术，而最好的架构一旦脱离了它所适应的场景，一切都将是空谈**。我希望你记住的是，这里所说的几点都只是一个个方向，你应该尽量往这些方向上去努力，但也要考虑平衡其他因素。

2.2 不同场景下的不同架构案例

前面我说了一些架构上的原则，那么针对“秒杀”这个场景，怎样才是一个好的架构呢？下面我以淘宝早期秒杀系统架构的演进为主线，来帮你梳理不同的请求体量下，我认为的最佳秒杀系统架构。

如果你想快速搭建一个简单的秒杀系统，只需要把你的商品购买页面增加一个“定时上架”功能，仅在秒杀开始时才让用户看到购买按钮，当商品的库存卖完了也就结束了。这就是当时第一个版本的秒杀系统实现方式。

但随着请求量的加大（比如从 1w/s 到了 10w/s 的量级），这个简单的架构很快就遇到了瓶颈，因此需要做架构改造来提升系统性能。这些架构改造包括：

1. 把秒杀系统独立出来单独打造一个系统，这样可以有针对性地做优化，例如这个独立出来的系统就减少了店铺装修的功能，减少了页面的复杂度；
2. 在系统部署上也独立做一个机器集群，这样秒杀的大流量就不会影响到正常的商品购买集群的机器负载；
3. 将热点数据（如库存数据）单独放到一个缓存系统中，以提高“读性能”；
4. 增加秒杀答题，防止有秒杀器抢单。

此时的系统架构变成了下图这个样子。最重要的就是，秒杀详情成为了一个独立的新系统，另外核心的一些数据放到了缓存（Cache）中，其他的关联系统也都以独立集群的方式进行部署。

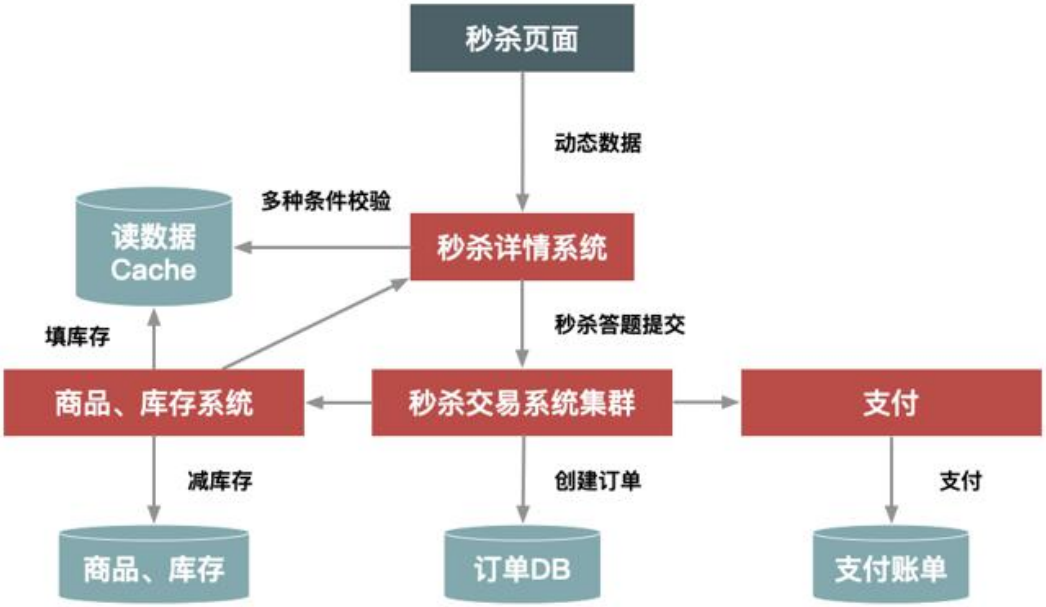

图 1 改造后的系统架构

然而这个架构仍然支持不了超过 100w/s 的请求量，所以为了进一步提升秒杀系统的性能，我们又对架构做进一步升级，比如：

1. 对页面进行彻底的动静分离，使得用户秒杀时不需要刷新整个页面，而只需要点击抢宝按钮，借此把页面刷新的数据降到最少；
2. 在服务端对秒杀商品进行本地缓存，不需要再调用依赖系统的后台服务获取数据，甚至不需要去公共的缓存集群中查询数据，这样不仅可以减少系统调用，而且能够避免压垮公共缓存集群。
3. 增加系统限流保护，防止最坏情况发生。

经过这些优化，系统架构变成了下图中的样子。在这里，我们对页面进行了进一步的静态化，秒杀过程中不需要刷新整个页面，而只需要向服务端请求很少的动态数据。而且，最关键的详情和交易系统都增加了本地缓存，来提前缓存秒杀商品的信息，热点数据库也做了独立部署，等等。

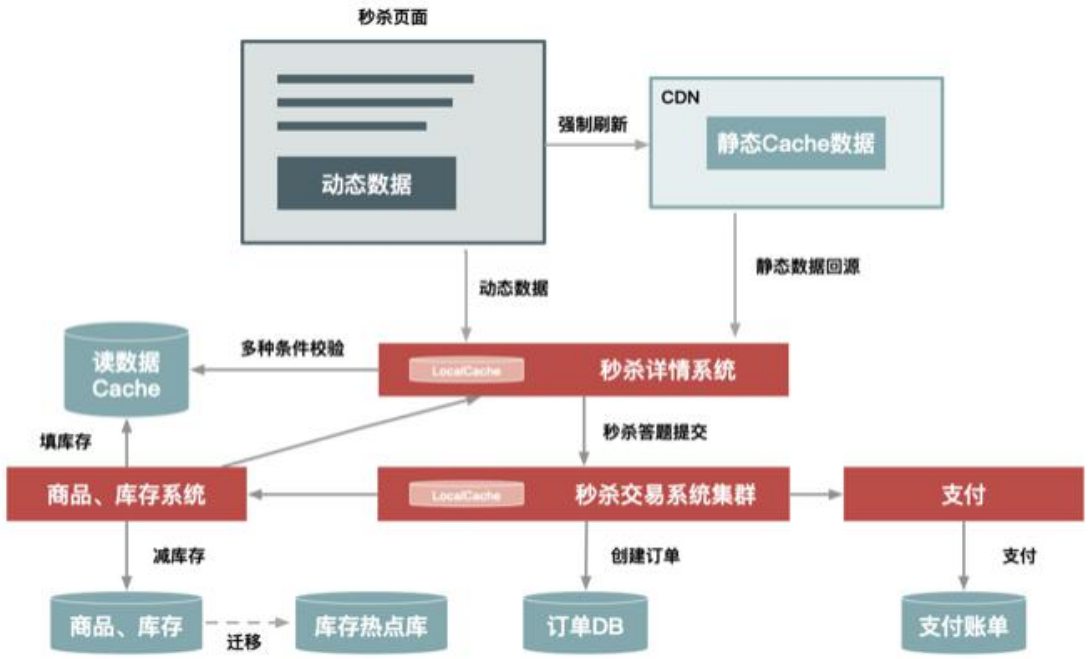

图 2 进一步改造后的系统架构

从前面的几次升级来看，其实越到后面需要定制的地方越多，也就是越“不通用”。例如，把秒杀商品缓存在每台机器的内存中，这种方式显然不适合太多的商品同时进行秒杀的情况，因为单机的内存始终有限。所以要取得极致的性能，就要在其他地方（比如，通用性、易用性、成本等方面）有所牺牲。

2.3. 总结

回顾下前面的内容，首先介绍了构建大并发、高性能、高可用系统中几种通用的优化思路，并抽象总结为“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。

然后，给出了实际构建秒杀系统时，根据不同级别的流量，由简单到复杂打造的几种系统架构，希望能供你参考。当然，这里面没有说具体的解决方案，比如缓存用什么、页面静态化用什么，因为这些对于架构来说并不重要，作为架构师，你应该时刻提醒自己主线是什么。

说了这么多，总体上我希望给你一个方向，就是想构建大并发、高性能、高可用的系统应该从哪几个方向上去努力，然后在不同性能要求的情况下系统架构应该

从哪几个方面去做取舍。同时你也要明白，越追求极致性能，系统定制开发就会越多，同时系统的通用性也就会越差。

三、如何才能做好动静分离？有哪些方案可选？

上一章，介绍了秒杀系统在架构上要考虑的几个原则，也许有人就想说：“知易行难，这些原则应该怎么应用到系统中呢？”别急，从这篇文章开始，我就会逐一介绍秒杀系统的各个关键环节中涉及的关键技术。

这一章就先来讨论第一个关键点：数据的动静分离。不知道你之前听过这个解决方案吗？不管你有没有听过，我都建议你先停下来思考动静分离的价值。如果你的系统还没有开始应用动静分离的方案，那你也可以想想为什么没有，是之前没有想到，还是说业务体量根本用不着？

不过我可以确信地说，如果你在一个业务飞速发展的公司里，并且你在深度参与公司内类秒杀类系统的架构或者开发工作，那么你迟早会想到动静分离的方案。为什么？很简单，**秒杀的场景中，对于系统的要求其实就三个字：快、准、稳**。

那怎么才能“快”起来呢？我觉得抽象起来讲，就只有两点，一点是提高单次请求的效率，一点是减少没必要的请求。今天我们聊到的“动静分离”其实就是瞄着这个大方向去的。

不知道你是否还记得，最早的秒杀系统其实是要刷新整体页面的，但后来秒杀的时候，你只要点击“刷新抢宝”按钮就够了，这种变化的本质就是动静分离，分离之后，客户端大幅度减少了请求的数据量。这不自然就“快”了吗？

3.1 何为动静数据

那到底什么才是动静分离呢？所谓“动静分离”，其实就是把用户请求的数据（如 HTML 页面）划分为“动态数据”和“静态数据”。

简单来说，**“动态数据”和“静态数据”的主要区别就是看页面中输出的数据是否和 URL、浏览者、时间、地域相关，以及是否含有 Cookie 等私密数据**。比如说：

1. 很多媒体类的网站，某一篇文章的内容不管是你访问还是我访问，它都是一样的。所以它就是一个典型的静态数据，但是它是个动态页面。

2. 我们如果现在访问淘宝的首页，每个人看到的页面可能都是不一样的，淘宝首页中包含了很多根据访问者特征推荐的信息，而这些个性化的数据就可以理解为动态数据了。

这里再强调一下，我们所说的静态数据，不能仅仅理解为传统意义上完全存在磁盘上的 HTML 页面，它也可能是经过 Java 系统产生的页面，但是它输出的页面本身不包含上面所说的那些因素。也就是所谓“动态”还是“静态”，并不是说数据本身是否动静，而是数据中是否含有和访问者相关的个性化数据。

还有一点要注意，就是页面中“不包含”，指的是“页面的 HTML 源码中不含有”，这一点务必要清楚。

理解了静态数据和动态数据，我估计你很容易就能想明白“动静分离”这个方案的来龙去脉了。分离了动静数据，我们就可以对分离出来的静态数据做缓存，有了缓存之后，静态数据的“访问效率”自然就提高了。

那么，怎样对静态数据做缓存呢？我在这里总结了几个重点。

第一，你应该把静态数据缓存到离用户最近的地方。静态数据就是那些相对不会变化的数据，因此我们可以把它们缓存起来。缓存到哪里呢？常见的就三种，用户浏览器里、CDN 上或者在服务端的 Cache 中。你应该根据情况，把它们尽量缓存到离用户最近的地方。

第二，静态化改造就是要直接缓存 HTTP 连接。相较于普通的数据缓存而言，你肯定还听过系统的静态化改造。静态化改造是直接缓存 HTTP 连接而不是仅仅缓存数据，如下图所示，Web 代理服务器根据请求 URL，直接取出对应的 HTTP 响应头和响应体然后直接返回，这个响应过程简单得连 HTTP 协议都不用重新组装，甚至连 HTTP 请求头也不需要解析。

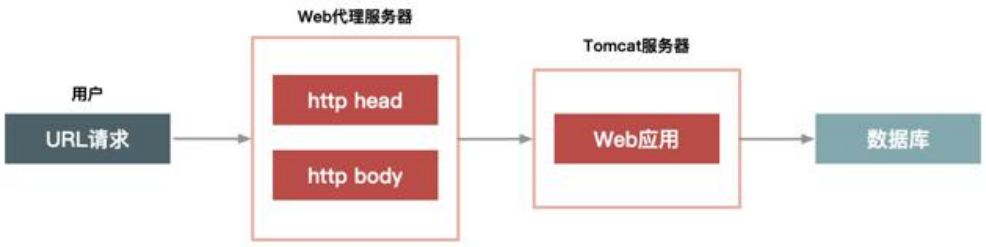

图 1 静态化改造

第三，让谁来缓存静态数据也很重要。不同语言写的 Cache 软件处理缓存数据的效率也各不相同。以 Java 为例，因为 Java 系统本身也有其弱点（比如不擅长处理大量连接请求，每个连接消耗的内存较多，Servlet 容器解析 HTTP 协议较慢），所以你可以不在 Java 层做缓存，而是直接在 Web 服务器层上做，这样你就可以屏蔽 Java 语言层面的一些弱点；而相比起来，Web 服务器（如 Nginx、Apache、Varnish）也更擅长处理大并发的静态文件请求。

3.2 如何做动静分离的改造

理解了动静态数据的“why”和“what”，接下来我们就要看“how”了。我们如何把动态页面改造成适合缓存的静态页面呢？其实也很简单，就是去除前面所说的那几个影响因素，把它们单独分离出来，做动静分离。

下面，我以典型的商品详情系统为例来详细介绍。这里，你可以先打开京东或者淘宝的商品详情页，看看这个页面里都有哪些动静数据。我们从以下 5 个方面来分离出动态内容。

1. **URL 唯一化**。商品详情系统天然地就可以做到 URL 唯一化，比如每个商品都由 ID 来标识，那么 http://item.xxx.com/item.htm?id=xxxx 就可以作为唯一的 URL 标识。为啥要 URL 唯一呢？前面说了我们是要缓存整个 HTTP 连接，那么以什么作为 Key 呢？就以 URL 作为缓存的 Key，例如以 id=xxx 这个格式进行区分。
2. **分离浏览者相关的因素**。浏览者相关的因素包括是否已登录，以及登录身份等，这些相关因素我们可以单独拆分出来，通过动态请求来获取。
3. **分离时间因素**。服务端输出的时间也通过动态请求获取。
4. **异步化地域因素**。详情页面上与地域相关的因素做成异步方式获取，当然你也可以通过动态请求方式获取，只是这里通过异步获取更合适。
5. **去掉 Cookie**。服务端输出的页面包含的 Cookie 可以通过代码软件来删除，如 Web 服务器 Varnish 可以通过 unset req.http.cookie 命令去掉 Cookie。注意，这里说的去掉 Cookie 并不是用户端收到的页面就不含 Cookie 了，而是说，在缓存的静态数据中不含有 Cookie。

分离出动态内容之后，如何组织这些内容页就变得非常关键了。这里我要提醒你一点，因为这其中很多动态内容都会被页面中的其他模块用到，如判断该用户是否已登录、用户 ID 是否匹配等，所以这个时候我们应该将这些信息 JSON 化（用 JSON 格式组织这些数据），以方便前端获取。

前面我们介绍里用缓存的方式来处理静态数据。而动态内容的处理通常有两种方案：ESI（Edge Side Includes）方案和 CSI（Client Side Include）方案。

1. **ESI 方案（或者 SSI）**：即在 Web 代理服务器上做动态内容请求，并将请求插入到静态页面中，当用户拿到页面时已经是一个完整的页面了。这种方式对服务端性能有些影响，但是用户体验较好。

2. CSI 方案。即单独发起一个异步 JavaScript 请求，以向服务端获取动态内容。这种方式服务端性能更佳，但是用户端页面可能会延时，体验稍差。

3.3 动静分离的几种架构方案

前面我们通过改造把静态数据和动态数据做了分离，那么如何在系统架构上进一步对这些动态和静态数据重新组合，再完整地输出给用户呢？

这就涉及对用户请求路径进行合理的架构了。根据架构上的复杂度，有 3 种方案可选：

1. 实体机单机部署；
2. 统一 Cache 层；
3. 上 CDN。

方案 1：实体机单机部署

这种方案是将虚拟机改为实体机，以增大 Cache 的容量，并且采用了一致性 Hash 分组的方式来提升命中率。这里将 Cache 分成若干组，是希望能达到命中率和访问热点的平衡。Hash 分组越少，缓存的命中率肯定就会越高，但短板是也会使单个商品集中在一个分组中，容易导致 Cache 被击穿，所以我们应该适当增加多个相同的分组，来平衡访问热点和命中率的问题。

这里我给出了实体机单机部署方案的结构图，如下：

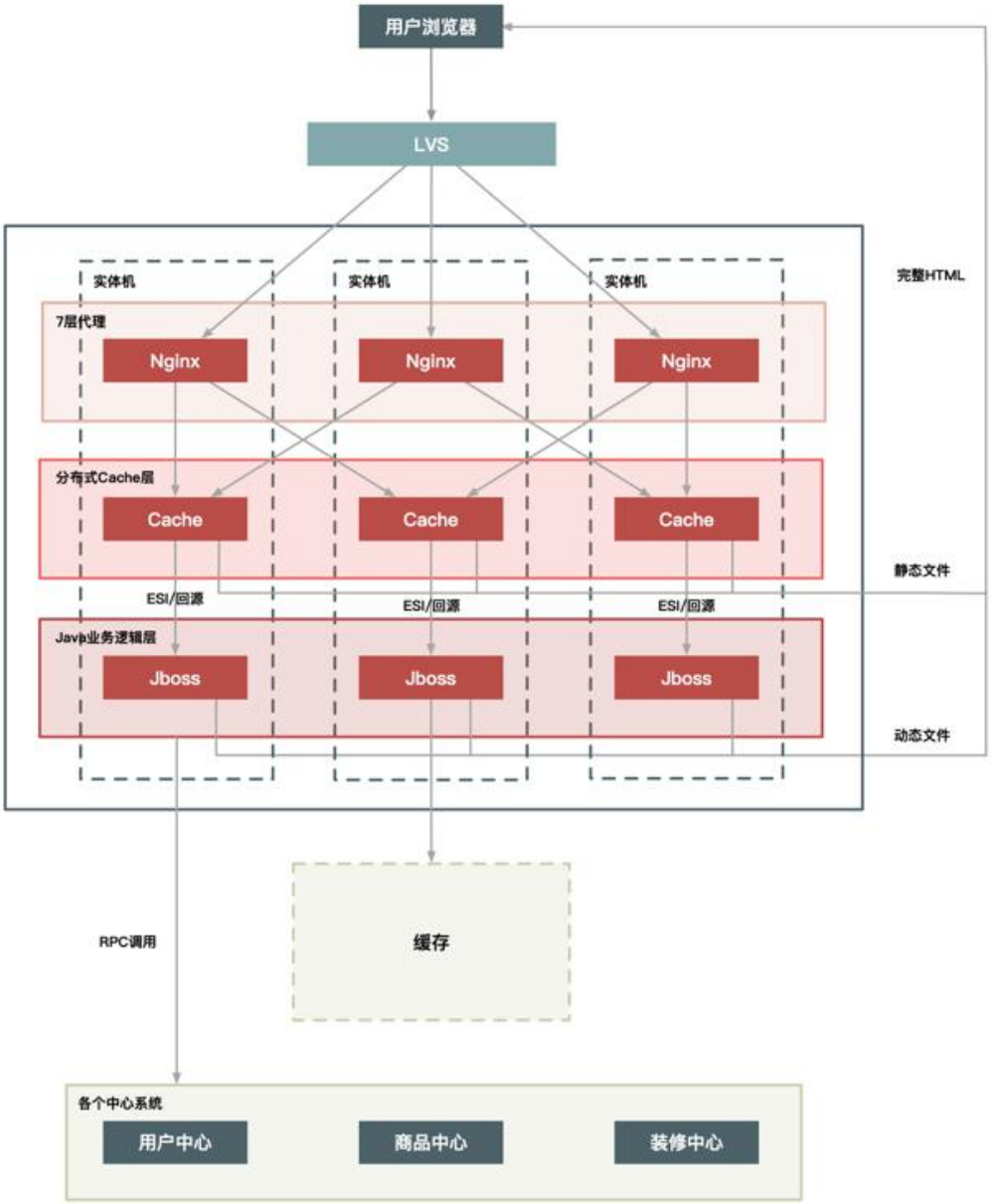

图 2 Nginx+Cache+Java 结构实体机单机部署

实体机单机部署有以下几个优点：

1. 没有网络瓶颈，而且能使用大内存；
2. 既能提升命中率，又能减少 Gzip 压缩；
3. 减少 Cache 失效压力，因为采用定时失效方式，例如只缓存 3 秒钟，过期即自动失效。

这个方案中，虽然把通常只需要虚拟机或者容器运行的 Java 应用换成实体机，优势很明显，它会增加单机的内存容量，但是一定程度上也造成了 CPU 的浪费，因为单个的 Java 进程很难用完整个实体机的 CPU。

另外就是，一个实体机上部署了 Java 应用又作为 Cache 来使用，这造成了运维上的高复杂度，所以这是一个折中的方案。如果你的公司里，没有更多的系统有类似需求，那么这样做也比较合适，如果你们有多个业务系统都有静态化改造的需求，那还是建议把 Cache 层单独抽出来公用比较合理，如下面的方案 2 所示。

方案 2：统一 Cache 层

所谓统一 Cache 层，就是将单机的 Cache 统一分离出来，形成一个单独的 Cache 集群。统一 Cache 层是个更理想的可推广方案，该方案的结构图如下：

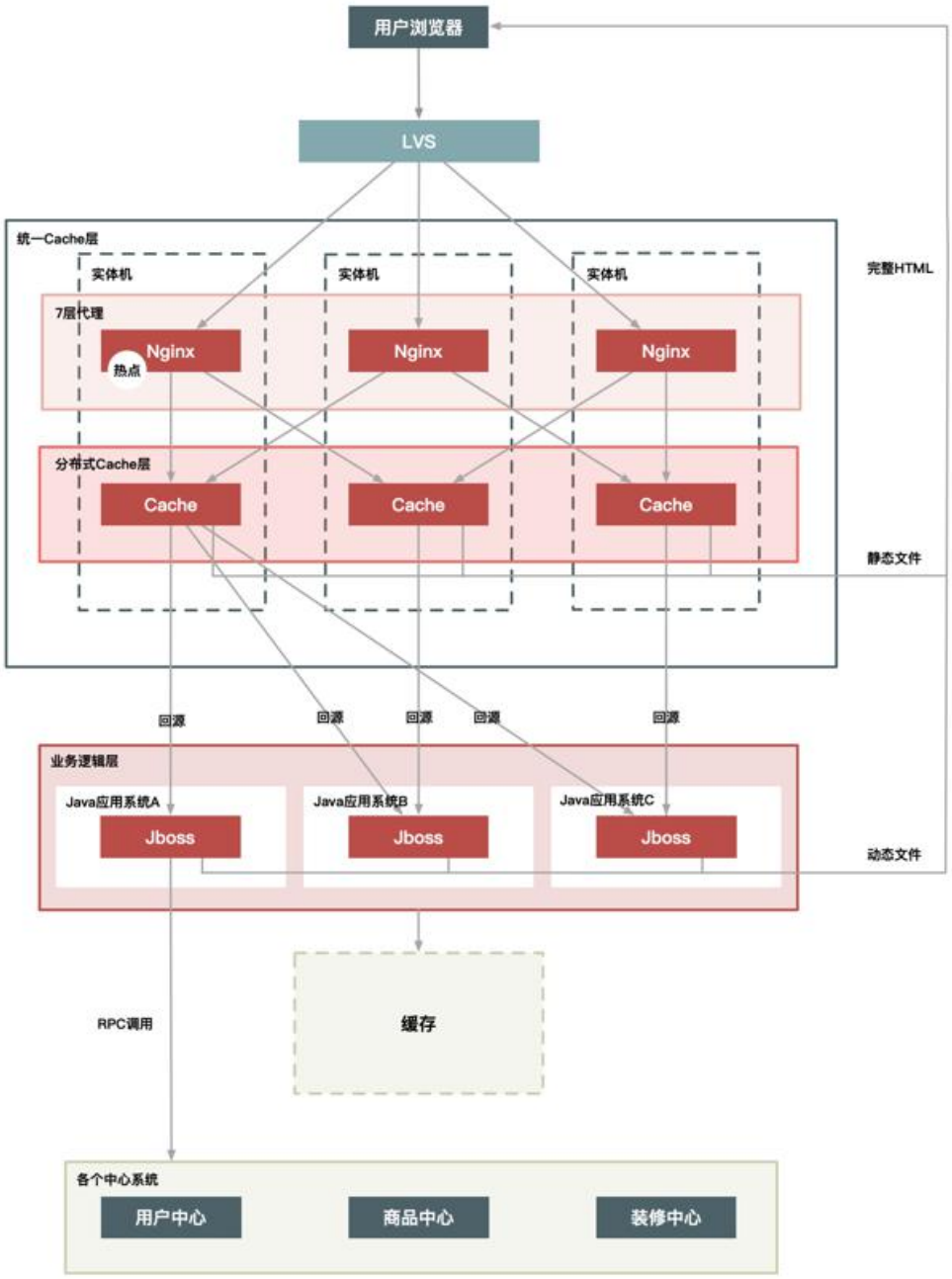

图 3 统一 Cache

将 Cache 层单独拿出来统一管理可以减少运维成本，同时也方便接入其他静态化系统。此外，它还有一些优点。

1. 单独一个 Cache 层，可以减少多个应用接入时使用 Cache 的成本。这样接入的应用只要维护自己的 Java 系统就好，不需要单独维护 Cache，而只关心如何使用即可。

2. 统一 Cache 的方案更易于维护，如后面加强监控、配置的自动化，只需要一套解决方案就行，统一起来维护升级也比较方便。
3. 可以共享内存，最大化利用内存，不同系统之间的内存可以动态切换，从而能够有效应对各种攻击。

这种方案虽然维护上更方便了，但是也带来了其他一些问题，比如缓存更加集中，导致：

1. Cache 层内部交换网络成为瓶颈；
2. 缓存服务器的网卡也会是瓶颈；
3. 机器少风险较大，挂掉一台就会影响很大一部分缓存数据。

要解决上面这些问题，可以再对 Cache 做 Hash 分组，即一组 Cache 缓存的内容相同，这样能够避免热点数据过度集中导致新的瓶颈产生。

方案 3：上 CDN

在将整个系统做动静分离后，我们自然会想到更进一步的方案，就是将 Cache 进一步前移到 CDN 上，因为 CDN 离用户最近，效果会更好。

但是要想这么做，有以下几个问题需要解决。

1. **失效问题**。前面我们也有提到过缓存时效的问题，不知道你有没有理解，我再来解释一下。谈到静态数据时，我说过一个关键词叫“相对不变”，它的言外之意是“可能会变化”。比如一篇文章，现在不变，但如果你发现个错别字，是不是就会变化了？如果你的缓存时效很长，那用户端在很长一段时间内看到的都是错的。所以，这个方案中也是，我们需要保证 CDN 可以在秒级时间内，让分布在全国各地的 Cache 同时失效，这对 CDN 的失效系统要求很高。
2. **命中率问题**。Cache 最重要的一个衡量指标就是“高命中率”，不然 Cache 的存在就失去了意义。同样，如果将数据全部放到全国的 CDN 上，必然导致 Cache 分散，而 Cache 分散又会导致访问请求命中同一个 Cache 的可能性降低，那么命中率就成为一个问题。
3. **发布更新问题**。如果一个业务系统每周都有日常业务需要发布，那么发布系统必须足够简洁高效，而且你还要考虑有问题时快速回滚和排查问题的简便性。

从前面的分析来看，将商品详情系统放到全国的所有 CDN 节点上是不太现实的，因为存在失效问题、命中率问题以及系统的发布更新问题。那么是否可以选择若干个节点来尝试实施呢？答案是“可以”，但是这样的节点需要满足几个条件：

1. 靠近访问量比较集中的地区；
2. 离主站相对较远；
3. 节点到主站间的网络比较好，而且稳定；

4. 节点容量比较大，不会占用其他 CDN 太多的资源。

最后，还有一点也很重要，那就是：节点不要太多。

基于上面几个因素，选择 CDN 的二级 Cache 比较合适，因为二级 Cache 数量偏少，容量也更大，让用户的请求先回源的 CDN 的二级 Cache 中，如果没命中再回源站获取数据，部署方式如下图所示：

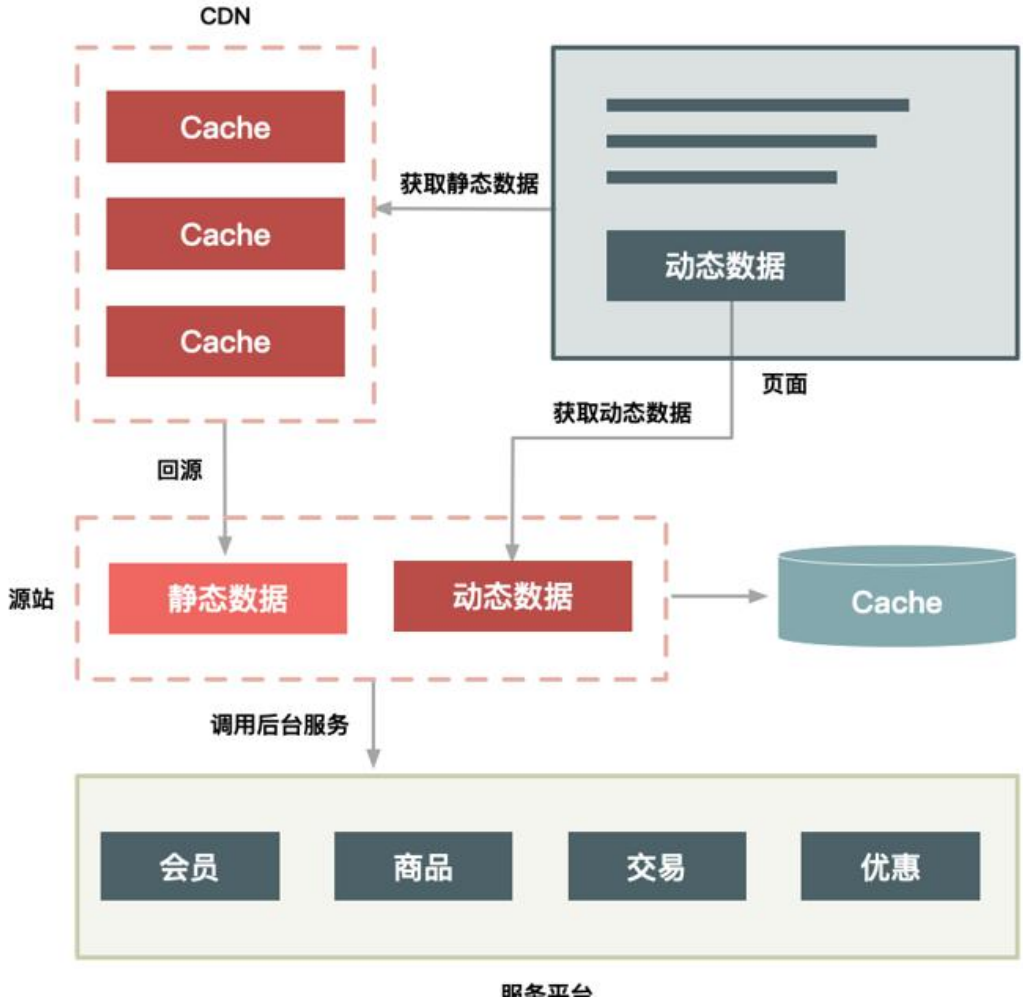

图 4 CDN 化部署方案

使用 CDN 的二级 Cache 作为缓存，可以达到和当前服务端静态化 Cache 类似的命中率，因为节点数不多，Cache 不是很分散，访问量也比较集中，这样也就解决了命中率问题，同时能够给用户最好的访问体验，是当前比较理想的一种 CDN 化方案。

除此之外，CDN 化部署方案还有以下几个特点：

1. 把整个页面缓存在用户浏览器中；
2. 如果强制刷新整个页面，也会请求 CDN；
3. 实际有效请求，只是用户对“刷新抢宝”按钮的点击。

这样就把 90% 的静态数据缓存在了用户端或者 CDN 上，当真正秒杀时，用户只需要点击特殊的“刷新抢宝”按钮，而不需要刷新整个页面。这样一来，系统只是向服务端请求很少的有效数据，而不需要重复请求大量的静态数据。

秒杀的动态数据和普通详情页面的动态数据相比更少，性能也提升了 3 倍以上。所以“抢宝”这种设计思路，让我们不用刷新页面就能够很好地请求到服务端最新的动态数据。

3.4 总结

这一章主要介绍了实现动静分离的几种思路，并由易到难给出了几种架构方案，以及它们各自的优缺点。可以看到，不同的架构方案会引入不同的问题，比如我们把缓存数据从 CDN 上移到用户的浏览器里，针对秒杀这个场景是没问题的，但针对一般的商品可否也这样做呢？

你可能会问，存储在浏览器或 CDN 上，有多大区别？我的回答是：区别很大！因为在 CDN 上，我们可以做主动失效，而在用户的浏览器里就更不可控，如果用户不主动刷新的话，你很难主动地把消息推送给用户的浏览器。

另外，在什么地方把静态数据和动态数据合并并渲染出一个完整的页面也很关键，假如在用户的浏览器里合并，那么服务端可以减少渲染整个页面的 CPU 消耗。如果在服务端合并的话，就要考虑缓存的数据是否进行 Gzip 压缩了：如果缓存 Gzip 压缩后的静态数据可以减少缓存的数据量，但是进行页面合并渲染时就要先解压，然后再压缩完整的页面数据输出给用户；如果缓存未压缩的静态数据，这样不用解压静态数据，但是会增加缓存容量。虽然这些都是细节问题，但你在设计架构方案时都需要考虑清楚。

四、二八原则：有针对性地处理好系统的“热点数据”

假设你的系统中存储有几十亿上百亿的商品，而每天有千万级的商品被上亿的用户访问，那么肯定有一部分被大量用户访问的热卖商品，这就是我们常说的“热点商品”。

这些热点商品中最极端的例子就是秒杀商品，它们在很短时间内被大量用户执行访问、添加购物车、下单等操作，这些操作我们就称为“热点操作”。那么问题来了：这些热点对系统有啥影响，我们非要关注这些热点吗？

4.1 为什么要关注热点

我们一定要关注热点，因为热点会对系统产生一系列的影响。

首先，热点请求会大量占用服务器处理资源，虽然这个热点可能只占请求总量的亿分之一，然而却可能抢占 90% 的服务器资源，如果这个热点请求还是没有价值的无效请求，那么对系统资源来说完全是浪费。

其次，即使这些热点是有效的请求，我们也要识别出来做针对性的优化，从而用更低的代价来支撑这些热点请求。

既然热点对系统来说这么重要，那么热点到底包含哪些内容呢？

4.2 什么是“热点”

热点分为**热点操作**和**热点数据**。所谓“热点操作”，例如大量的刷新页面、大量的添加购物车、双十一零点大量的下单等都属于此类操作。对系统来说，这些操作可以抽象为“读请求”和“写请求”，这两种热点请求的处理方式大相径庭，读请求的优化空间要大一些，而写请求的瓶颈一般都在存储层，优化的思路就是根据 CAP 理论做平衡，这个内容我在“减库存”一文再详细介绍。

而“热点数据”比较好理解，那就是用户的热点请求对应的数据。而热点数据又分为“静态热点数据”和“动态热点数据”。

所谓“静态热点数据”，就是能够提前预测的热点数据。例如，我们可以通过卖家报名的方式提前筛选出来，通过报名系统对这些热点商品进行打标。另外，我们还可以通过大数据分析来提前发现热点商品，比如我们分析历史成交记录、用户的购物车记录，来发现哪些商品可能更热门、更好卖，这些都是可以提前分析出来的热点。

所谓“动态热点数据”，就是不能被提前预测到的，系统在运行过程中临时产生的热点。例如，卖家在抖音上做了广告，然后商品一下就火了，导致它在短时间内被大量购买。

由于热点操作是用户的行为，我们不好改变，但能做一些限制和保护，所以本文我主要针对热点数据来介绍如何进行优化。

4.3 发现热点数据

前面，我介绍了如何对单个秒杀商品的页面数据进行动静分离，以便针对性地对静态数据做优化处理，那么另外一个关键的问题来了：如何发现这些秒杀商品，或者更准确地说，如何发现热点商品呢？

你可能会说“参加秒杀的商品就是秒杀商品啊”，没错，关键是系统怎么知道哪些商品参加了秒杀活动呢？所以，你要有一个机制提前来区分普通商品和秒杀商品。

我们从发现静态热点和发现动态热点两个方面来看一下。

发现静态热点数据

如前面讲的，静态热点数据可以通过商业手段，例如强制让卖家通过报名参加的方式提前把热点商品筛选出来，实现方式是通过一个运营系统，把参加活动的商品数据进行打标，然后通过一个后台系统对这些热点商品进行预处理，如提前进行缓存。但是这种通过报名提前筛选的方式也会带来新的问题，即增加卖家的使用成本，而且实时性较差，也不太灵活。

不过，除了提前报名筛选这种方式，你还可以通过技术手段提前预测，例如对买家每天访问的商品进行大数据计算，然后统计出 TOP N 的商品，我们可以认为这些 TOP N 的商品就是热点商品。

发现动态热点数据

我们可以通过卖家报名或者大数据预测这些手段来提前预测静态热点数据，但这其中有一个痛点，就是实时性较差，如果我们的系统能在秒级内自动发现热点商品那就完美了。

能够动态地实时发现热点不仅对秒杀商品，对其他热卖商品也同样有价值，所以我们需要想办法实现热点的动态发现功能。

这里我给出一个动态热点发现系统的具体实现。

1. 构建一个异步的系统，它可以收集交易链路上各个环节中的中间件产品的热点 Key，如 Nginx、缓存、RPC 服务框架等这些中间件（一些中间件产品本身已经有热点统计模块）。
2. 建立一个热点上报和可以按照需求订阅的热点服务的下发规范，主要目的是通过交易链路上各个系统（包括详情、购物车、交易、优惠、库存、物流等）访问的时间差，把上游已经发现的热点透传给下游系统，提前做好保护。

比如，对于大促高峰期，详情系统是最早知道的，在统一接入层上 Nginx 模块统计的热点 URL。

3. 将上游系统收集的热点数据发送到热点服务台，然后下游系统（如交易系统）就会知道哪些商品会被频繁调用，然后做热点保护。

这里我给出了一个图，其中用户访问商品时经过的路径有很多，我们主要是依赖前面的导购页面（包括首页、搜索页面、商品详情、购物车等）提前识别哪些商品的访问量高，通过这些系统中的中间件来收集热点数据，并记录到日志中。

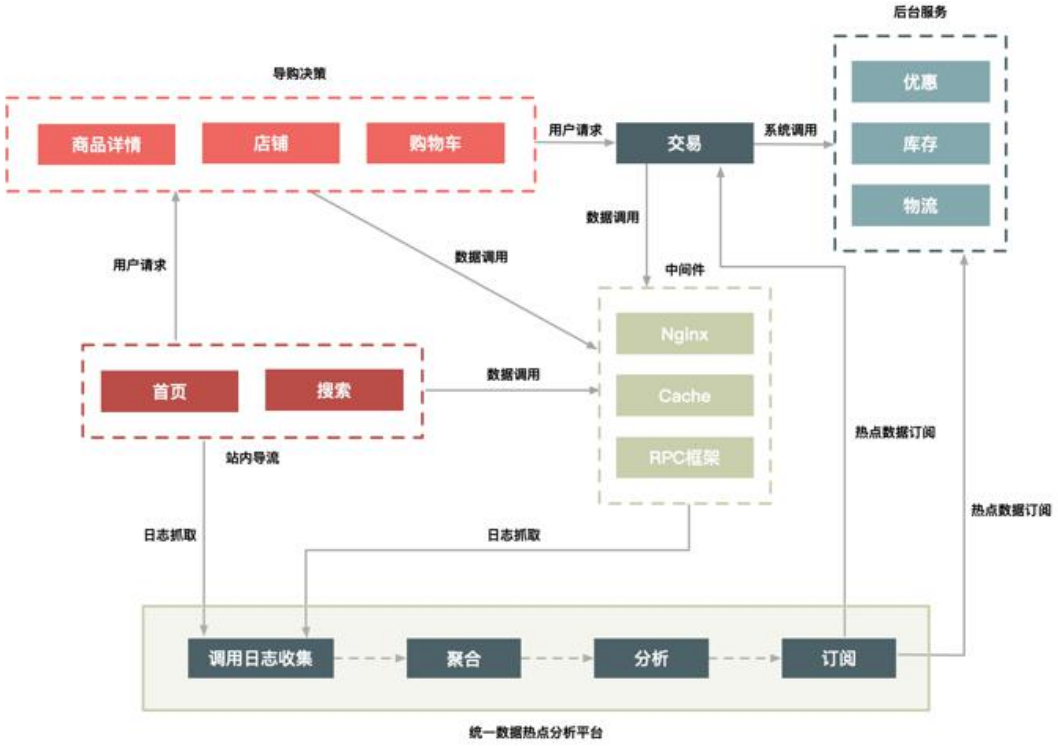

图 1 一个动态热点发现系统

我们通过部署在每台机器上的 Agent 把日志汇总到聚合和分析集群中，然后把符合一定规则的热点数据，通过订阅分发系统再推送到相应的系统中。你可以是把热点数据填充到 Cache 中，或者直接推送到应用服务器的内存中，还可以对这些数据进行拦截，总之下游系统可以订阅这些数据，然后根据自己的需求决定如何处理这些数据。

打造热点发现系统时，我根据以往经验总结了几点注意事项。

1. 这个热点服务后台抓取热点数据日志最好采用异步方式，因为“异步”一方面便于保证通用性，另一方面又不影响业务系统和中间件产品的主流程。

2. 热点服务发现和中间件自身的热点保护模块并存，每个中间件和应用还需要保护自己。热点服务台提供热点数据的收集和订阅服务，便于把各个系统的热点数据透明出来。
3. 热点发现要做到接近实时（3s 内完成热点数据的发现），因为只有做到接近实时，动态发现才有意义，才能实时地对下游系统提供保护。

4.4 处理热点数据

处理热点数据通常有几种思路：一是优化，二是限制，三是隔离。

先来说说优化。优化热点数据最有效的办法就是缓存热点数据，如果热点数据做了动静分离，那么可以长期缓存静态数据。但是，缓存热点数据更多的是“临时”缓存，即不管是静态数据还是动态数据，都用一个队列短暂地缓存数秒钟，由于队列长度有限，可以采用 LRU 淘汰算法替换。

再来说说限制。限制更多的是一种保护机制，限制的办法也有很多，例如对被访问商品的 ID 做一致性 Hash，然后根据 Hash 做分桶，每个分桶设置一个处理队列，这样可以把热点商品限制在一个请求队列里，防止因某些热点商品占用太多的服务器资源，而使其他请求始终得不到服务器的处理资源。

最后介绍一下隔离。秒杀系统设计的第一个原则就是将这种热点数据隔离出来，不要让 1% 的请求影响到另外的 99%，隔离出来后也更方便对这 1% 的请求做针对性的优化。

具体到“秒杀”业务，我们可以在以下几个层次实现隔离。

1. **业务隔离**。把秒杀做成一种营销活动，卖家要参加秒杀这种营销活动需要单独报名，从技术上来说，卖家报名后对我们来说就有了已知热点，因此可以提前做好预热。
2. **系统隔离**。系统隔离更多的是运行时的隔离，可以通过分组部署的方式和另外 99% 分开。秒杀可以申请单独的域名，目的也是让请求落到不同的集群中。
3. **数据隔离**。秒杀所调用的数据大部分都是热点数据，比如会启用单独的 Cache 集群或者 MySQL 数据库来放热点数据，目的也是不想 0.01% 的数据有机会影响 99.99% 数据。

当然了，实现隔离有很多种办法。比如，你可以按照用户来区分，给不同的用户分配不同的 Cookie，在接入层，路由到不同的服务接口中；再比如，你还可以在接入层针对 URL 中的不同 Path 来设置限流策略。服务层调用不同的服务接口，以及数据层通过给数据打标来区分等等这些措施，其目的都是把已经识别出来的热点请求和普通的请求区分开。

4.5 总结

本章与数据的动静分离不一样，它从另外一个维度对数据进行了区分处理。你要明白，区分的目的主要还是对读热点数据加以优化，对照“4 要 1 不要”原则，它可以减少请求量，也可以减少请求的路径。因为缓存的数据都是经过多个请求，或者从多个系统中获取的数据经过计算后的结果。

热点的发现和隔离不仅对“秒杀”这个场景有意义，对其他的高性能分布式系统也非常有价值，尤其是热点的隔离非常重要。我介绍了业务层面的隔离和数据层面的隔离方式，最重要最简单的方式就是独立出来一个集群，单独处理热点数据。

但是能够独立出来一个集群的前提还是首先能够发现热点，为此我介绍了发现热点的几种方式，比如人工标识、大数据统计计算，以及实时热点发现方案，希望能够给你启发。

五、流量削峰这事应该怎么做？

如果你看过秒杀系统的流量监控图的话，你会发现它是一条直线，就在秒杀开始那一秒是一条很直很直的线，这是因为秒杀请求在时间上高度集中于某一特定的时间点。这样一来，就会导致一个特别高的流量峰值，它对资源的消耗是瞬时的。

但是对秒杀这个场景来说，最终能够抢到商品的人数是固定的，也就是说 100 人和 10000 人发起请求的结果都是一样的，并发度越高，无效请求也越多。

但是从业务上来说，秒杀活动是希望更多的人来参与的，也就是开始之前希望有更多的人来刷页面，但是真正开始下单时，秒杀请求并不是越多越好。因此我们可以设计一些规则，让并发的请求更多地延缓，而且我们甚至可以过滤掉一些无效请求。

5.1 为什么要削峰

为什么要削峰呢？或者说峰值会带来哪些坏处？

我们知道服务器的处理资源是恒定的，你用或者不用它的处理能力都是一样的，所以出现峰值的话，很容易导致忙到处理不过来，闲的时候却又没有什么要处理。但是由于要保证服务质量，我们的很多处理资源只能按照忙的时候来预估，而这会导致资源的一个浪费。

这就好比因为存在早高峰和晚高峰的问题，所以有了错峰限行的解决方案。削峰的存在，一是可以让服务端处理变得更加平稳，二是可以节省服务器的资源成本。

针对秒杀这一场景，削峰从本质上来说就是更多地延缓用户请求的发出，以便减少和过滤掉一些无效请求，它遵从“请求数要尽量少”的原则。

下面就来介绍一下流量削峰的一些操作思路：排队、答题、分层过滤。这几种方式都是无损（即不会损失用户的发出请求）的实现方案，当然还有些有损的实现方案，包括我们后面要介绍的关于稳定性的一些办法，比如限流和机器负载保护等一些强制措施也能达到削峰保护的目的，当然这都是不得已的一些措施，因此就不归类到这里了。

5.2 排队

要对流量进行削峰，最容易想到的解决方案就是用消息队列来缓冲瞬时流量，把同步的直接调用转换成异步的间接推送，中间通过一个队列在一端承接瞬时的流量洪峰，在另一端平滑地将消息推送出去。在这里，消息队列就像“水库”一样，拦蓄上游的洪水，削减进入下游河道的洪峰流量，从而达到减免洪水灾害的目的。

用消息队列来缓冲瞬时流量的方案，如下图所示：

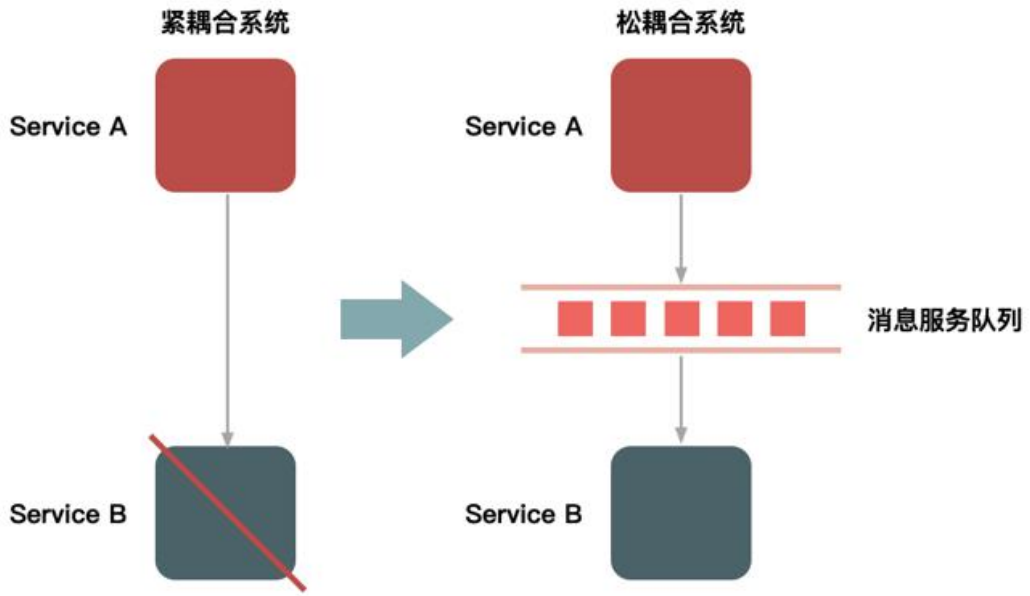

图 1 用消息队列来缓冲瞬时流量

但是，如果流量峰值持续一段时间达到了消息队列的处理上限，例如本机的消息积压达到了存储空间的上限，消息队列同样也会被压垮，这样虽然保护了下游的系统，但是和直接把请求丢弃也没多大的区别。就像遇到洪水爆发时，即使是有水库恐怕也无济于事。

除了消息队列，类似的排队方式还有很多，例如：

1. 利用线程池加锁等待也是一种常用的排队方式；
2. 先进先出、先进后出等常用的内存排队算法的实现方式；
3. 把请求序列化到文件中，然后再顺序地读文件（例如基于 MySQL binlog 的同步机制）来恢复请求等方式。

可以看到，这些方式都有一个共同特征，就是把“一步的操作”变成“两步的操作”，其中增加的一步操作用来起到缓冲的作用。

说到这里你可能会说，这样一来增加了访问请求的路径啊，并不符合我们介绍的“4 要 1 不要”原则。没错，的确看起来不太合理，但是如果不增加一个缓冲步骤，那么在一些场景下系统很可能会直接崩溃，所以最终还是需要你做出妥协和平衡。

5.3 答题

你是否还记得，最早期的秒杀只是纯粹地刷新页面和点击购买按钮，它是后来才增加了答题功能的。那么，为什么要增加答题功能呢？

这主要是为了增加购买的复杂度，从而达到两个目的。

第一个目的是防止部分买家使用秒杀器在参加秒杀时作弊。2011 年秒杀非常火的时候，秒杀器也比较猖獗，因而没有达到全民参与和营销的目的，所以系统增加了答题来限制秒杀器。增加答题后，下单的时间基本控制在 2s 后，秒杀器的下单比例也大大下降。答题页面如下图所示。

图 2 答题页面

第二个目的其实就是延缓请求，起到对请求流量进行削峰的作用，从而让系统能够更好地支持瞬时的流量高峰。这个重要的功能就是把峰值的下单请求拉长，从

以前的 1s 之内延长到 2s~10s。这样一来，请求峰值基于时间分片了。这个时间的分片对服务端处理并发非常重要，会大大减轻压力。而且，由于请求具有先后顺序，靠后的请求到来时自然也就没有库存了，因此根本到不了最后的下单步骤，所以真正的并发写就非常有限了。这种设计思路目前用得非常普遍，如当年支付宝的“咻一咻”、微信的“摇一摇”都是类似的方式。

这里，我重点说一下秒杀答题的设计思路。

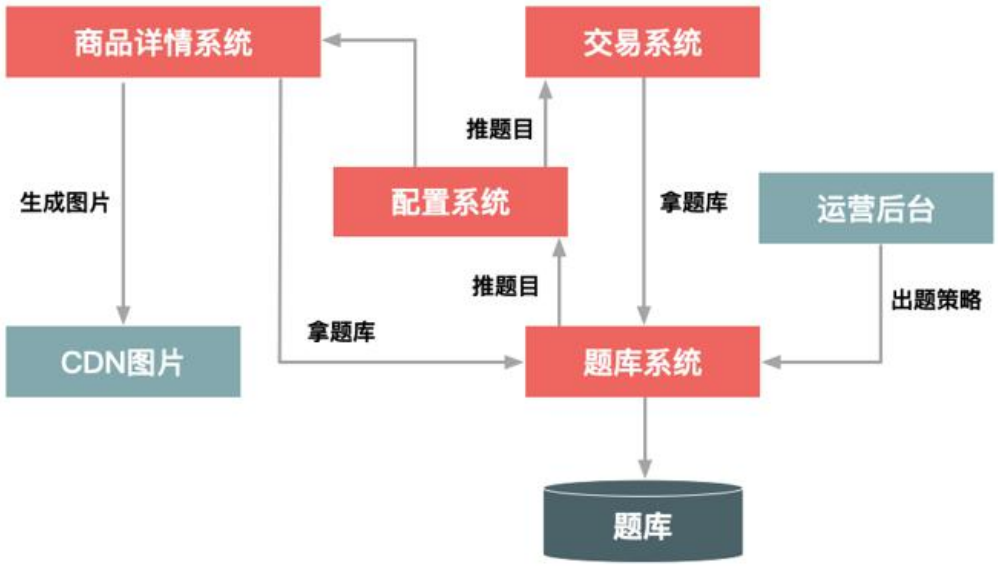

图 3 秒杀答题

如上图所示，整个秒杀答题的逻辑主要分为 3 部分。

1. **题库生成模块**，这个部分主要就是生成一个个问题和答案，其实题目和答案本身并不需要很复杂，重要的是能够防止由机器来算出结果，即防止秒杀器来答题。
2. **题库的推送模块**，用于在秒杀答题前，把题目提前推送给详情系统和交易系统。题库的推送主要是为了保证每次用户请求的题目是唯一的，目的也是防止答题作弊。
3. **题目的图片生成模块**，用于把题目生成为图片格式，并且在图片里增加一些干扰因素。这也同样是为防止机器直接来答题，它要求只有人才能理解题目本身的含义。这里还要注意一点，由于答题时网络比较拥挤，我们应该把题目的图片提前推送到 CDN 上并且要进行预热，不然的话当用户真正请求题目时，图片可能加载比较慢，从而影响答题的体验。

其实真正答题的逻辑比较简单，很好理解：当用户提交的答案和题目对应的答案做比较，如果通过了就继续进行下一步的下单逻辑，否则就失败。我们可以把问题和答案用下面这样的 key 来进行 MD5 加密：

问题 key：userId+itemId+question_Id+time+PK

答案 key：userId+itemId+answer+PK

验证的逻辑如下图所示：

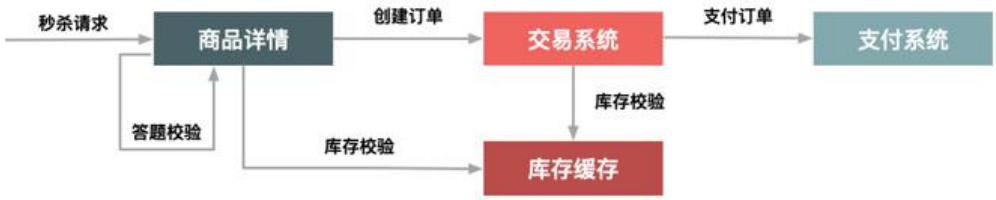

图 4 答题的验证逻辑

注意，这里面的验证逻辑，除了验证问题的答案以外，还包括用户本身身份的验证，例如是否已经登录、用户的 Cookie 是否完整、用户是否重复频繁提交等。

除了做正确性验证，我们还可以对提交答案的时间做些限制，例如从开始答题到接受答案要超过 1s，因为小于 1s 是人为操作的可能性很小，这样也能防止机器答题的情况。

5.4 分层过滤

前面介绍的排队和答题要么是少发请求，要么对发出来的请求进行缓冲，而针对秒杀场景还有一种方法，就是对请求进行分层过滤，从而过滤掉一些无效的请求。分层过滤其实就是采用“漏斗”式设计来处理请求的，如下图所示。

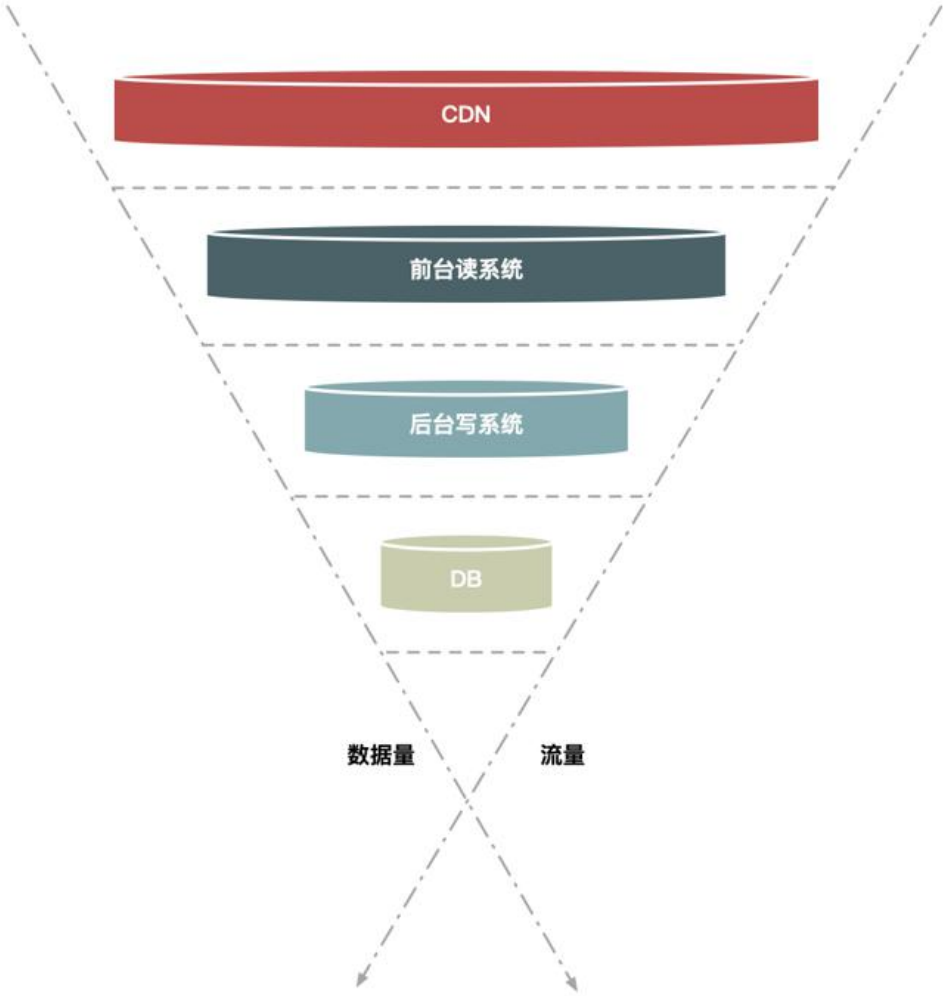

图 5 分层过滤

假如请求分别经过 CDN、前台读系统（如商品详情系统）、后台系统（如交易系统）和数据库这几层，那么：

- 大部分数据和流量在用户浏览器或者 CDN 上获取，这一层可以拦截大部分数据的读取；
- 经过第二层(即前台系统)时数据(包括强一致性的数据)尽量得走 Cache，过滤一些无效的请求；
- 再到第三层后台系统，主要做数据的二次检验，对系统做好保护和限流，这样数据量和请求就进一步减少；
- 最后在数据层完成数据的强一致性校验。

这样就像漏斗一样，尽量把数据量和请求量一层一层地过滤和减少了。

分层过滤的核心思想是：在不同的层次尽可能地过滤掉无效请求，让“漏斗”最末端的才是有效请求。而要达到这种效果，我们就必须对数据做分层的校验。

分层校验的基本原则是：

1. 将动态请求的读数据缓存（Cache）在 Web 端，过滤掉无效的数据读；
2. 对读数据不做强一致性校验，减少因为一致性校验产生瓶颈的问题；
3. 对写数据进行基于时间的合理分片，过滤掉过期的失效请求；
4. 对写请求做限流保护，将超出系统承载能力的请求过滤掉；
5. 对写数据进行强一致性校验，只保留最后有效的数据。

分层校验的目的是：在读系统中，尽量减少由于一致性校验带来的系统瓶颈，但是尽量将不影响性能的检查条件提前，如用户是否具有秒杀资格、商品状态是否正常、用户答题是否正确、秒杀是否已经结束、是否非法请求、营销等价物是否充足等；在写数据系统中，主要对写的数据（如“库存”）做一致性检查，最后在数据库层保证数据的最终准确性（如“库存”不能减为负数）。

5.5 总结

本章介绍了如何在网站面临大流量冲击时进行请求的削峰，并主要介绍了削峰的 3 种处理方式：一个是通过队列来缓冲请求，即控制请求的发出；一个是通过答题来延长请求发出的时间，在请求发出后承接请求时进行控制，最后再对不符合条件的请求进行过滤；最后一种是对请求进行分层过滤。

其中，队列缓冲方式更加通用，它适用于内部上下游系统之间调用请求不平缓的场景，由于内部系统的服务质量要求不能随意丢弃请求，所以使用消息队列能起到很好的削峰和缓冲作用。

而答题更适用于秒杀或者营销活动等应用场景，在请求发起端就控制发起请求的速度，因为越到后面无效请求也会越多，所以配合后面介绍的分层拦截的方式，可以更进一步减少无效请求对系统资源的消耗。

分层过滤非常适合交易性的写请求，比如减库存或者拼车这种场景，在读的时候需要知道还有没有库存或者是否还有剩余空座位。但是由于库存和座位又是不停变化的，所以读的数据是否一定要非常准确呢？其实不一定，你可以放一些请求过去，然后在真正减的时候再做强一致性保证，这样既过滤一些请求又解决了强一致性读的瓶颈。

不过，在削峰的处理方式上除了采用技术手段，其实还可以采用业务手段来达到一定效果，例如在零点开启大促的时候由于流量太大导致支付系统阻塞，这个时候可以采用发放优惠券、发起抽奖活动等方式，将一部分流量分散到其他地方，这样也能起到缓冲流量的作用。

六、影响性能的因素有哪些？又该如何提高系统的性能？

看到这，你对于秒杀系统的构建有没有形成一些框架性的认识，这里再带你简单回忆下前面的主线。

前面的章节，介绍的内容多少都和优化有关：第一篇介绍了一些指导原则；第二篇和第三篇从动静分离和热点数据两个维度，介绍了如何有针对性地对数据进行区分和优化处理；第四章介绍了在保证实现基本业务功能的前提下，尽量减少和过滤一些无效请求的思路。

这几章既是在讲根据指导原则实现的具体案例，也是在讲如何实现能够让整个系统更“快”。我想说的是，优化本身有很多手段，也是一个复杂的系统工程。本章就来结合秒杀这一场景，重点给介绍下服务端的一些优化技巧。

6.1 影响性能的因素

你想要提升性能，首先肯定要知道哪些因素对于系统性能的影响最大，然后再针对这些具体的因素想办法做优化，是不是这个逻辑？

那么，哪些因素对性能有影响呢？在回答这个问题之前，我们先定义一下“性能”，服务设备不同对性能的定义也是不一样的，例如 CPU 主要看主频、磁盘主要看 IOPS（Input/Output Operations Per Second，即每秒进行读写操作的次数）。

而今天我们讨论的主要是系统服务端性能，一般用 QPS（Query Per Second，每秒请求数）来衡量，还有一个影响和 QPS 也息息相关，那就是响应时间（Response Time，RT），它可以理解为服务器处理响应的耗时。

正常情况下响应时间（RT）越短，一秒钟处理的请求数（QPS）自然也就会越多，这在单线程处理的情况下看起来是线性的关系，即我们只要把每个请求的响应时间降到最低，那么性能就会最高。

但是你可能想到响应时间总有一个极限，不可能无限下降，所以又出现了另外一个维度，即通过多线程，来处理请求。这样理论上就变成了“总 QPS =（1000ms / 响应时间）× 线程数量”，这样性能就和两个因素相关了，一个是一次响应的服务端耗时，一个是处理请求的线程数。

接下来，我们一起看看这个两个因素到底会造成什么样的影响。

首先，我们先来看看响应时间和 QPS 有啥关系。

对于大部分的 Web 系统而言，响应时间一般都是由 CPU 执行时间和线程等待时间（比如 RPC、IO 等待、Sleep、Wait 等）组成，即服务器在处理一个请求时，一部分是 CPU 本身在做运算，还有一部分是在各种等待。

理解了服务器处理请求的逻辑，估计你会说为什么我们不去减少这种等待时间。很遗憾，根据我们实际的测试发现，减少线程等待时间对提升性能的影响没有我们想象得那么大，它并不是线性的提升关系，这点在很多代理服务器（Proxy）上可以做验证。

如果代理服务器本身没有 CPU 消耗，我们在每次给代理服务器代理的请求加个延时，即增加响应时间，但是这对代理服务器本身的吞吐量并没有多大的影响，因为代理服务器本身的资源并没有被消耗，可以通过增加代理服务器的处理线程数，来弥补响应时间对代理服务器的 QPS 的影响。

其实，真正对性能有影响的是 CPU 的执行时间。这也很好理解，因为 CPU 的执行真正消耗了服务器的资源。经过实际的测试，如果减少 CPU 一半的执行时间，就可以增加一倍的 QPS。

也就是说，我们应该致力于减少 CPU 的执行时间。

其次，我们再来看看线程数对 QPS 的影响。

单看“总 QPS”的计算公式，你会觉得线程数越多 QPS 也就会越高，但这会一直正确吗？显然不是，线程数不是越多越好，因为线程本身也消耗资源，也受到其他因素的制约。例如，线程越多系统的线程切换成本就会越高，而且每个线程也都会耗费一定内存。

那么，设置什么样的线程数最合理呢？其实**很多多线程的场景都有一个默认配置，即“线程数 = 2 * CPU 核数 + 1”**。除去这个配置，还有一个根据最佳实践得出来的公式：

线程数 = [(线程等待时间 + 线程 CPU 时间) / 线程 CPU 时间] × CPU 数量

当然，最好的办法是通过性能测试来发现最佳的线程数。

换句话说，要提升性能我们就要减少 CPU 的执行时间，另外就是要设置一个合理的并发线程数，通过这两方面来显著提升服务器的性能。

现在，你知道了如何来快速提升性能，那接下来你估计会问，我应该怎么发现系统哪里最消耗 CPU 资源呢？

6.2 如何发现瓶颈

就服务器而言，会出现瓶颈的地方有很多，例如 CPU、内存、磁盘以及网络等都可能会导致瓶颈。此外，不同的系统对瓶颈的关注度也不一样，例如对缓存系统而言，制约它的是内存，而对存储型系统来说 I/O 更容易是瓶颈。

我们定位的场景是秒杀，它的瓶颈更多地发生在 CPU 上。

那么，如何发现 CPU 的瓶颈呢？其实有很多 CPU 诊断工具可以发现 CPU 的消耗，最常用的就是 JProfiler 和 Yourkit 这两个工具，它们可以列出整个请求中每个函数的 CPU 执行时间，可以发现哪个函数消耗的 CPU 时间最多，以便你有针对性地做优化。

当然还有一些办法也可以近似地统计 CPU 的耗时，例如通过 jstack 定时地打印调用栈，如果某些函数调用频繁或者耗时较多，那么那些函数就会多次出现在系统调用栈里，这样相当于采样的方式也能够发现耗时较多的函数。

虽说秒杀系统的瓶颈大部分在 CPU，但这并不表示其他方面就一定不出现瓶颈。例如，如果海量请求涌过来，你的页面又比较大，那么网络就有可能出现瓶颈。

怎样简单地判断 CPU 是不是瓶颈呢？一个办法就是看当 QPS 达到极限时，你的服务器的 CPU 使用率是不是超过了 95%，如果没有超过，那么表示 CPU 还有提升的空间，要么是有锁限制，要么是有过多的本地 I/O 等待发生。

现在你知道了优化哪些因素，又发现了瓶颈，那么接下来就要关注如何优化了。

6.3 如何优化系统

对 Java 系统来说，可以优化的地方很多，这里我重点说一下比较有效的几种手段，供你参考，它们是：减少编码、减少序列化、Java 极致优化、并发读优化。接下来，我们分别来看一下。

6.3.1. 减少编码

Java 的编码运行比较慢，这是 Java 的一大硬伤。在很多场景下，只要涉及字符串的操作（如输入输出操作、I/O 操作）都比较耗 CPU 资源，不管它是磁盘 I/O 还是网络 I/O，因为都需要将字符转换成字节，而这个转换必须编码。

每个字符的编码都需要查表，而这种查表的操作非常耗资源，所以减少字符到字节或者相反的转换、减少字符编码会非常有成效。减少编码就可以大大提升性能。

那么如何才能减少编码呢？例如，网页输出是可以直接进行流输出的，即用 resp.getOutputStream() 函数写数据，把一些静态的数据提前转化成字节，等到真正往外写的时候再直接用 OutputStream() 函数写，就可以减少静态数据的编码转换。

我在《深入分析 Java Web 技术内幕》一书中介绍的“Velocity 优化实践”一章的内容，就是基于把静态的字符串提前编码成字节并缓存，然后直接输出字节内容到页面，从而大大减少编码的性能消耗的，网页输出的性能比没有提前进行字符到字节转换时提升了 30% 左右。

6.3.2. 减少序列化

序列化也是 Java 性能的一大天敌，减少 Java 中的序列化操作也能大大提升性能。又因为序列化往往是和编码同时发生的，所以减少序列化也就减少了编码。

序列化大部分是在 RPC 中发生的，因此避免或者减少 RPC 就可以减少序列化，当然当前的序列化协议也已经做了很多优化来提升性能。有一种新的方案，就是可以将多个关联性比较强的应用进行“合并部署”，而减少不同应用之间的 RPC 也可以减少序列化的消耗。

所谓“合并部署”，就是把两个原本在不同机器上的不同应用合并部署到一台机器上，当然不仅仅是部署在一台机器上，还要在同一个 Tomcat 容器中，且不能走本机的 Socket，这样才能避免序列化的产生。

另外针对秒杀场景，我们还可以做得更极致一些，接下来我们来看第 3 点：Java 极致优化。

6.3.3. Java 极致优化

Java 和通用的 Web 服务器（如 Nginx 或 Apache 服务器）相比，在处理大并发的 HTTP 请求时要弱一点，所以一般我们都会对大流量的 Web 系统做静态化改造，让大部分请求和数据直接在 Nginx 服务器或者 Web 代理服务器（如 Varnish、Squid 等）上直接返回（这样可以减少数据的序列化与反序列化），而 Java 层只需处理少量数据的动态请求。针对这些请求，我们可以使用以下手段进行优化：

直接使用 Servlet 处理请求。避免使用传统的 MVC 框架，这样可以绕过一大堆复杂且用处不大的处理逻辑，节省 1ms 时间（具体取决于你对 MVC 框架的依赖程度）。

直接输出流数据。使用 resp.getOutputStream() 而不是 resp.getWriter() 函数，可以省掉一些不变字符数据的编码，从而提升性能；数据输出时推荐使用 JSON 而不是模板引擎（一般都是解释执行）来输出页面。

6.3.4. 并发读优化

也许有读者会觉得这个问题很容易解决，无非就是放到 Tair 缓存里面。集中式缓存为了保证命中率一般都会采用一致性 Hash，所以同一个 key 会落到同一台机器上。虽然单台缓存机器也能支撑 30w/s 的请求，但还是远不足以应对像“大秒”这种级别的热点商品。那么，该如何彻底解决单点的瓶颈呢？

答案是采用应用层的 LocalCache，即在秒杀系统的单机上缓存商品相关的数据。

那么，又如何缓存（Cache）数据呢？你需要划分成动态数据和静态数据分别进行处理：

- 像商品中的“标题”和“描述”这些本身不变的数据，会在秒杀开始之前全量推送到秒杀机器上，并一直缓存到秒杀结束；
- 像库存这类动态数据，会采用“被动失效”的方式缓存一定时间（一般是数秒），失效后再去缓存拉取最新的数据。

你可能还会有疑问：像库存这种频繁更新的数据，一旦数据不一致，会不会导致超卖？

这就要用到前面介绍的读数据的分层校验原则了，读的场景可以允许一定的脏数据，因为这里的误判只会导致少量原本无库存的下单请求被误认为有库存，可以等到真正写数据时再保证最终的一致性，通过在数据的高可用性和一致性之间的平衡，来解决高并发的数据读取问题。

6.4 总结

性能优化的过程首先要从发现短板开始，除了我今天介绍的一些优化措施外，你还可以在减少数据、数据分级（动静分离），以及减少中间环节、增加预处理等这些环节上做优化。

首先是“发现短板”，比如考虑以下因素的一些限制：光速（光速：C = 30 万千米 / 秒；光纤：V = C/1.5=20 万千米 / 秒，即数据传输是有物理距离的限制的）、网速（2017 年 11 月知名测速网站 Ookla 发布报告，全国平均上网带宽达到 61.24 Mbps，千兆带宽下 10KB 数据的极限 QPS 为 1.25 万 QPS=1000Mbps/8/10KB）、网络结构（交换机 / 网卡的限制）、TCP/IP、虚拟机（内存 /CPU/IO 等资源的限制）和应用本身的一些瓶颈等。

其次是减少数据。事实上，有两个地方特别影响性能，一是服务端在处理数据时不可避免地存在字符到字节的相互转化，二是 HTTP 请求时要做 Gzip 压缩，还有网络传输的耗时，这些都和数据大小密切相关。

再次，就是数据分级，也就是要保证首屏为先、重要信息为先，次要信息则异步加载，以这种方式提升用户获取数据的体验。

最后就是要减少中间环节，减少字符到字节的转换，增加预处理（提前做字符到字节的转换）去掉不需要的操作。

此外，要做好优化，你还需要做好应用基线，比如性能基线（何时性能突然下降）、成本基线（去年双 11 用了多少台机器）、链路基线（我们的系统发生了哪些变化），你可以通过这些基线持续关注系统的性能，做到在代码上提升编码质量，在业务上改掉不合理的调用，在架构和调用链路上不断的改进。

七、秒杀系统“减库存”设计的核心逻辑

如果要设计一套秒杀系统，那我想你的老板肯定会先对你说：千万不要超卖，这是大前提。

如果你第一次接触秒杀，那你可能还不太理解，库存 100 件就卖 100 件，在数据库里减到 0 就好了啊，这有什么麻烦的？是的，理论上是这样，但是具体到业务场景中，“减库存”就不是这么简单了。

例如，我们平常购物都是这样，看到喜欢的商品然后下单，但并不是每个下单请求你都最后付款了。你说系统是用户下单了就算这个商品卖出去了，还是等到用户真正付款了才算卖出了呢？这的确是个问题！

我们可以先根据减库存是发生在下单阶段还是付款阶段，把减库存做一下划分。

7.1 减库存有哪几种方式

在正常的电商平台购物场景中，用户的实际购买过程一般分为两步：下单和付款。你想买一台 iPhone 手机，在商品页面点了“立即购买”按钮，核对信息之后点击“提交订单”，这一步称为下单操作。下单之后，你只有真正完成付款操作才能算真正购买，也就是俗话说的“落袋为安”。

那如果你是架构师，你会在哪个环节完成减库存的操作呢？总结来说，减库存操作一般有如下几个方式：

下单减库存，即当买家下单后，在商品的总库存中减去买家购买数量。下单减库存是最简单的减库存方式，也是控制最精确的一种，下单时直接通过数据库的事务机制控制商品库存，这样一定不会出现超卖的情况。但是你要知道，有些人下完单可能并不会付款。

付款减库存，即买家下单后，并不立即减库存，而是等到有用户付款后才真正减库存，否则库存一直保留给其他买家。但因为付款时才减库存，如果并发比较高，有可能出现买家下单后付不了款的情况，因为可能商品已经被其他人买走了。

预扣库存，这种方式相对复杂一些，买家下单后，库存为其保留一定的时间（如 10 分钟），超过这个时间，库存将会自动释放，释放后其他买家就可以继续购买。在买家付款前，系统会校验该订单的库存是否还有保留：如果没有保留，则再次尝试预扣；如果库存不足（也就是预扣失败）则不允许继续付款；如果预扣成功，则完成付款并实际地减去库存。

以上这几种减库存的方式都会存在一些问题，下面我们一起来看下。

7.2 减库存可能存在的问题

由于购物过程中存在两步或者多步的操作，因此在不同的操作步骤中减库存，就会存在一些可能被恶意买家利用的漏洞，例如发生恶意下单的情况。

假如我们采用“下单减库存”的方式，即用户下单后就减去库存，正常情况下，买家下单后付款的概率会很高，所以不会有太大问题。但是有一种场景例外，就是当卖家参加某个活动时，此时活动的有效时间是商品的黄金售卖时间，如果有竞争对手通过恶意下单的方式将该卖家的商品全部下单，让这款商品的库存减为零，那么这款商品就不能正常售卖了。要知道，这些恶意下单的人是不会真正付款的，这正是“下单减库存”方式的不足之处。

既然“下单减库存”可能导致恶意下单，从而影响卖家的商品销售，那么有没有办法解决呢？你可能会想，采用“付款减库存”的方式是不是就可以了？的确可以。但是，“付款减库存”又会导致另外一个问题：库存超卖。

假如有 100 件商品，就可能出现 300 人下单成功的情况，因为下单时不会减库存，所以也就可能出现下单成功数远远超过真正库存数的情况，这尤其会发生在做活动的热门商品上。这样一来，就会导致很多买家下单成功但是付不了款，买家的购物体验自然比较差。

可以看到，不管是“下单减库存”还是“付款减库存”，都会导致商品库存不能完全和实际售卖情况对应起来的情况，看来要把商品准确地卖出去还真是不容易啊！

那么，既然“下单减库存”和“付款减库存”都有缺点，我们能否把两者相结合，将两次操作进行前后关联起来，下单时先预扣，在规定时间内不付款再释放库存，即采用“预扣库存”这种方式呢？

这种方案确实可以在一定程度上缓解上面的问题。但是否就彻底解决了呢？其实没有！针对恶意下单这种情况，虽然把有效的付款时间设置为 10 分钟，但是恶意买家完全可以在 10 分钟后再次下单，或者采用一次下单很多件的方式把库存减完。针对这种情况，解决办法还是要结合安全和反作弊的措施来制止。

例如，给经常下单不付款的买家进行识别打标（可以在被打标的买家下单时不减库存）、给某些类目设置最大购买件数（例如，参加活动的商品一人最多只能买 3 件），以及对重复下单不付款的操作进行次数限制等。

针对“库存超卖”这种情况，在 10 分钟时间内下单的数量仍然有可能超过库存数量，遇到这种情况我们只能区别对待：对普通的商品下单数量超过库存数量的情况，可以通过补货来解决；但是有些卖家完全不允许库存为负数的情况，那只能在买家付款时提示库存不足。

7.3 大型秒杀中如何减库存？

目前来看，业务系统中最常见的就是预扣库存方案，像你在买机票、买电影票时，下单后一般都有个“有效付款时间”，超过这个时间订单自动释放，这都是典型的预扣库存方案。而具体到秒杀这个场景，应该采用哪种方案比较好呢？

由于参加秒杀的商品，一般都是“抢到就是赚到”，所以成功下单后却不付款的情况比较少，再加上卖家对秒杀商品的库存有严格限制，所以秒杀商品采用“下单减库存”更加合理。另外，理论上由于“下单减库存”比“预扣库存”以及涉及第三方支付的“付款减库存”在逻辑上更为简单，所以性能上更占优势。

“下单减库存”在数据一致性上，主要就是保证大并发请求时库存数据不能为负数，也就是要保证数据库中的库存字段值不能为负数，一般我们有多种解决方案：一种是在应用程序中通过事务来判断，即保证减后库存不能为负数，否则就回滚；另一种办法是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；再有一种就是使用 CASE WHEN 判断语句，例如这样的 SQL 语句：

```
UPDATE item SET inventory = CASE WHEN inventory >= xxx THEN inventory-xxx ELSE inventory END
```

7.4 秒杀减库存的极致优化

在交易环节中，“库存”是个关键数据，也是个热点数据，因为交易的各个环节中都可能涉及对库存的查询。但是，我在前面介绍分层过滤时提到过，秒杀中并不需要对库存有精确的一致性读，把库存数据放到缓存（Cache）中，可以大大提升读性能。

解决大并发读问题，可以采用 LocalCache（即在秒杀系统的单机上缓存商品相关的数据）和对数据进行分层过滤的方式，但是像减库存这种大并发写无论如何还是避免不了，这也是秒杀场景下最为核心的一个技术难题。

因此，这里我想专门来说一下秒杀场景下减库存的极致优化思路，包括如何在缓存中减库存以及如何在数据库中减库存。

秒杀商品和普通商品的减库存还是有些差异的，例如商品数量比较少，交易时间段也比较短，因此这里有一个大胆的假设，即能否把秒杀商品减库存直接放到缓存系统中实现，也就是直接在缓存中减库存或者在一个带有持久化功能的缓存系统（如 Redis）中完成呢？

如果你的秒杀商品的减库存逻辑非常单一，比如没有复杂的 SKU 库存和总库存这种联动关系的话，我觉得完全可以。但是如果有比较复杂的减库存逻辑，或者需要使用事务，你还是必须在数据库中完成减库存。

由于 MySQL 存储数据的特点，同一数据在数据库里肯定是一行存储（MySQL），因此会有大量线程来竞争 InnoDB 行锁，而并发度越高时等待线程会越多，TPS（Transaction Per Second，即每秒处理的消息数）会下降，响应时间（RT）会上升，数据库的吞吐量就会严重受影响。

这就可能引发一个问题，就是单个热点商品会影响整个数据库的性能， 导致 0.01% 的商品影响 99.99% 的商品的售卖，这是我们不愿意看到的情况。一个解决思路是遵循前面介绍的原则进行隔离，把热点商品放到单独的热点库中。但是这无疑会带来维护上的麻烦，比如要做热点数据的动态迁移以及单独的数据库等。

而分离热点商品到单独的数据库还是没有解决并发锁的问题，我们应该怎么办呢？要解决并发锁的问题，有两种办法：

- **应用层做排队**。按照商品维度设置队列顺序执行，这样能减少同一台机器对数据库同一行记录进行操作的并发度，同时也能控制单个商品占用数据库连接的数量，防止热点商品占用太多的数据库连接。
- **数据库层做排队**。应用层只能做到单机的排队，但是应用机器数本身很多，这种排队方式控制并发的能力仍然有限，所以如果能在数据库层做全局排队是最

理想的。阿里的数据库团队开发了针对这种 MySQL 的 InnoDB 层上的补丁程序（patch），可以在数据库层上对单行记录做到并发排队。

你可能有疑问了，排队和锁竞争不都是要等待吗，有啥区别？

如果熟悉 MySQL 的话，你会知道 InnoDB 内部的死锁检测，以及 MySQL Server 和 InnoDB 的切换会比较消耗性能，淘宝的 MySQL 核心团队还做了很多其他方面的优化，如 COMMIT_ON_SUCCESS 和 ROLLBACK_ON_FAIL 的补丁程序，配合在 SQL 里面加提示（hint），在事务里不需要等待应用层提交（COMMIT），而在数据执行完最后一条 SQL 后，直接根据 TARGET_AFFECT_ROW 的结果进行提交或回滚，可以减少网络等待时间（平均约 0.7ms）。据我所知，目前阿里 MySQL 团队已经将包含这些补丁程序的 MySQL 开源。

另外，数据更新问题除了前面介绍的热点隔离和排队处理之外，还有些场景（如对商品的 lastmodifytime 字段的）更新会非常频繁，在某些场景下这些多条 SQL 是可以合并的，一定时间内只要执行最后一条 SQL 就行了，以便减少对数据库的更新操作。

7.5 总结

今天，我围绕商品减库存的场景，介绍了减库存的三种实现方案，以及分别存在的问题和可能的缓解办法。最后，我又聚焦秒杀这个场景说了如何实现减库存，以及在这个场景下做到极致优化的一些思路。

当然减库存还有很多细节问题，例如预扣的库存超时后如何进行库存回补，再比如目前都是第三方支付，如何在付款时保证减库存和成功付款时的状态一致性，这些都是很大的挑战。

八、如何设计兜底方案?

前面看了很多极致的优化思路，以及架构的优化方案。但是很遗憾，现实中总难免会发生一些这样或者那样的意外，而这些看似不经意的意外，却可能带来非常严重的后果。

我想对于很多秒杀系统而言，在诸如双十一这样的大流量的迅猛冲击下，都曾经或多或少发生过宕机的情况。当一个系统面临持续的大流量时，它其实很难单靠自身调整来恢复状态，你必须等待流量自然下降或者人为地把流量切走才行，这无疑会严重影响用户的购物体验。

同时，你也要知道，没有人能够提前预估所有情况，意外无法避免。那么，我们是不是就没办法了呢？当然不是，我们可以在系统达到不可用状态之前就做好流量限制，防止最坏情况的发生。用现在流行的话来说，任何一个系统，都需要“反脆弱”。

具体到秒杀这一场景下，为了保证系统的高可用，我们必须设计一个 Plan B 方案来兜底，这样在最坏情况发生时我们仍然能够从容应对。今天，我们就来看下兜底方案设计的一些具体思路。

8.1 高可用建设应该从哪里着手

说到系统的高可用建设，它其实是一个系统工程，需要考虑到系统建设的各个阶段，也就是说它其实贯穿了系统建设的整个生命周期，如下图所示：

图 1 高可用系统建设

具体来说，系统的高可用建设涉及架构阶段、编码阶段、测试阶段、发布阶段、运行阶段，以及故障发生时。接下来，我们分别看一下。

1. **架构阶段**：架构阶段主要考虑系统的可扩展性和容错性，要避免系统出现单点问题。例如多机房单元化部署，即使某个城市的某个机房出现整体故障，仍然不会影响整体网站的运转。
2. **编码阶段**：编码最重要的是保证代码的健壮性，例如涉及远程调用问题时，要设置合理的超时退出机制，防止被其他系统拖垮，也要对调用的返回结果集有预期，防止返回的结果超出程序处理范围，最常见的做法就是对错误异常进行捕获，对无法预料的错误要有默认处理结果。
3. **测试阶段**：测试主要是保证测试用例的覆盖度，保证最坏情况发生时，我们也有相应的处理流程。
4. **发布阶段**：发布时也有一些地方需要注意，因为发布时最容易出现错误，因此要有紧急的回滚机制。

5. **运行阶段**：运行时是系统的常态，系统大部分时间都会处于运行态，运行态最重要的是对系统的监控要准确及时，发现问题能够准确报警并且报警数据要准确详细，以便于排查问题。
6. **故障发生**：故障发生时首先最重要的就是及时止损，例如由于程序问题导致商品价格错误，那就要及时下架商品或者关闭购买链接，防止造成重大资产损失。然后就是要能够及时恢复服务，并定位原因解决问题。

为什么系统的高可用建设要放到整个生命周期中全面考虑？因为我们在每个环节中都可能犯错，而有些环节犯的错，你在后面是无法弥补的。例如在架构阶段，你没有消除单点问题，那么系统上线后，遇到突发流量把单点给挂了，你就只能干瞪眼，有时候想加机器都加不进去。所以高可用建设是一个系统工程，必须在每个环节都做好。

那么针对秒杀系统，我们重点介绍在遇到大流量时，应该从哪些方面来保障系统的稳定运行，所以更多的是看如何针对运行阶段进行处理，这就引出了接下来的内容：降级、限流和拒绝服务。

8.2 降级

所谓“降级”，就是当系统的容量达到一定程度时，限制或者关闭系统的某些非核心功能，从而把有限的资源保留给更核心的业务。它是一个有目的、有计划的执行过程，所以对降级我们一般需要有一套预案来配合执行。如果我们把它系统化，就可以通过预案系统和开关系统来实现降级。

降级方案可以这样设计：当秒杀流量达到 5w/s 时，把成交记录的获取从展示 20 条降级到只展示 5 条。“从 20 改到 5”这个操作由一个开关来实现，也就是设置一个能够从开关系统动态获取的系统参数。

这里，我给出开关系统的示意图。它分为两部分，一部分是开关控制台，它保存了开关的具体配置信息，以及具体执行开关所对应的机器列表；另一部分是执行下发开关数据的 Agent，主要任务就是保证开关被正确执行，即使系统重启后也会生效。

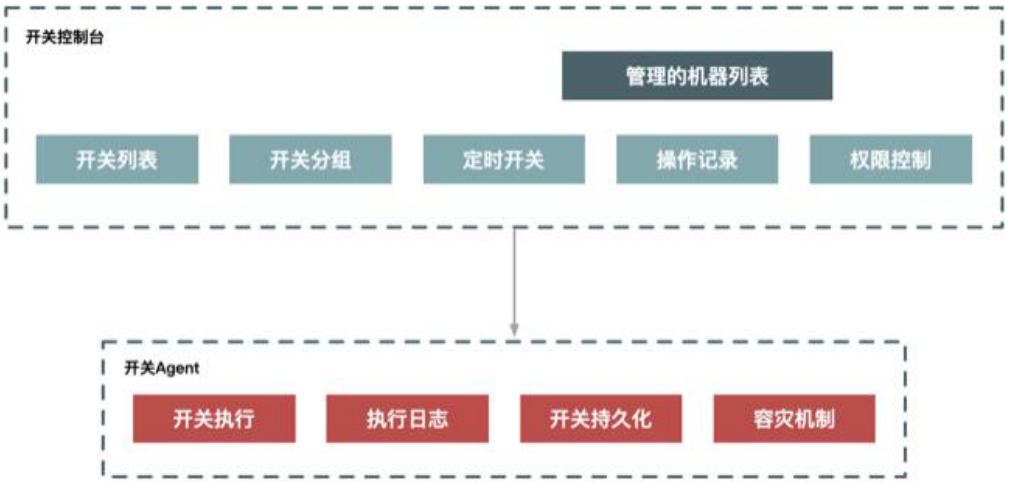

图 2 开关系统

执行降级无疑是在系统性能和用户体验之间选择了前者，降级后肯定会影响一部分用户的体验，例如在双 11 零点时，如果优惠券系统扛不住，可能会临时降级商品详情的优惠信息展示，把有限的系统资源用在保障交易系统正确展示优惠信息上，即保障用户真正下单时的价格是正确的。所以降级的核心目标是牺牲次要的功能和用户体验来保证核心业务流程的稳定，是一个不得已而为之的举措。

8.3 限流

如果说降级是牺牲了一部分次要的功能和用户的体验效果，那么限流就是更极端的一种保护措施了。限流就是当系统容量达到瓶颈时，我们需要通过限制一部分流量来保护系统，并做到既可以人工执行开关，也支持自动化保护的措施。

这里，我同样给出了限流系统的示意图。总体来说，限流既可以是在客户端限流，也可以是在服务端限流。此外，限流的实现方式既要支持 URL 以及方法级别的限流，也要支持基于 QPS 和线程的限流。

首先，我以内部的系统调用为例，来分别说下客户端限流和服务端限流的优缺点。

客户端限流，好处可以限制请求的发出，通过减少发出无用请求从而减少对系统的消耗。缺点就是当客户端比较分散时，没法设置合理的限流阈值：如果阈值设的太小，会导致服务端没有达到瓶颈时客户端已经被限制；而如果设的太大，则起不到限制的作用。

服务端限流，好处是可以根据服务端的性能设置合理的阈值，而缺点就是被限制的请求都是无效的请求，处理这些无效的请求本身也会消耗服务器资源。

图 3 限流系统

在限流的实现手段上来讲，基于 QPS 和线程数的限流应用最多，最大 QPS 很容易通过压测提前获取，例如我们的系统最高支持 1w QPS 时，可以设置 8000 来进行限流保护。线程数限流在客户端比较有效，例如在远程调用时我们设置连接池的线程数，超出这个并发线程请求，就将线程进行排队或者直接超时丢弃。

限流无疑会影响用户的正常请求，所以必然会导致一部分用户请求失败，因此在系统处理这种异常时一定要设置超时时间，防止因被限流的请求不能 fast fail（快速失败）而拖垮系统。

8.4 拒绝服务

如果限流还不能解决问题，最后一招就是直接拒绝服务了。

当系统负载达到一定阈值时，例如 CPU 使用率达到 90% 或者系统 load 值达到 2*CPU 核数时，系统直接拒绝所有请求，这种方式是最暴力但也最有效的系统保护方式。例如秒杀系统，我们在如下几个环节设计过载保护：

> 在最前端的 Nginx 上设置过载保护，当机器负载达到某个值时直接拒绝 HTTP 请求并返回 503 错误码，在 Java 层同样也可以设计过载保护。

拒绝服务可以说是一种不得已的兜底方案，用以防止最坏情况发生，防止因把服务器压跨而长时间彻底无法提供服务。像这种系统过载保护虽然在过载时无法提

供服务，但是系统仍然可以运作，当负载下降时又很容易恢复，所以每个系统和每个环节都应该设置这个兜底方案，对系统做最坏情况下的保护。

8.5 总结

网站的高可用建设是基础，可以说要深入到各个环节，更要长期规划并进行体系化建设，要在预防（建立常态的压力体系，例如上线前的单机压测到上线后的全链路压测）、管控（做好线上运行时的降级、限流和兜底保护）、监控（建立性能基线来记录性能的变化趋势以及线上机器的负载报警体系，发现问题及时预警）和恢复体系（遇到故障要及时止损，并提供快速的数据订正工具等）等这些地方加强建设，每一个环节可能都有很多事情要做。

另外，要保证高可用建设的落实，你不仅要做系统建设，还要在组织上做好保障。高可用其实就是在说“稳定性”。稳定性是一个平时不重要，但真出了问题就会要命的事儿，所以很可能平时业务发展良好，稳定性建设就会给业务让路，相关的稳定性负责人员平时根本得不到重视，一旦遇到故障却又成了“背锅侠”。

而要防止出现这种情况，就必须在组织上有所保障，例如可以让业务负责人背上稳定性 KPI 考核指标，然后在技术部门中建立稳定性建设小组，小组成员由每个业务线的核心力量兼任，他们的 KPI 由稳定性负责人来打分，这样稳定性小组就可以把一些体系化的建设任务落实到具体的业务系统中了。