16 | 秒杀场景：热点扣减如何保证命中的存储分片不挂?

从“第 12 讲”到“第 14 讲”，我们介绍了可以应对百万并发扣减请求，以及同时能够保障高性能的架构方案。此外，上述的架构方案还具备水平扩展能力，即当流量增加后，可以通过扩容底层存储和应用服务器来应对。

但面对百万并发的极端场景，比如大量用户在同一时间内抢购同一商品，前几讲介绍的几种方案就不能有效地应对了。此外，我们在“第 06 讲”里，介绍过热点查询的应对方案，是否可以直接复制来应对热点扣减呢？答案显然是不能的。

因此，在本讲里，我将先向你介绍**热点扣减的业务特点**，以及它**与热点查询的区别**，然后再循序渐进地介绍热点扣减的有效应对方案。

热点扣减的典型业务场景

热点扣减有一个被大家熟知的名称，叫作**秒杀**。其实，**秒杀并不等同于热点扣减**，只是因为商品秒杀是热点扣减里最具有代表性、也最能体现热点扣减特点的场景，所以我们常常以秒杀代指热点扣减。秒杀的特点主要有以下两点。

1. 首先，**秒杀带来的热点量非常大**，其他热点场景很难比拟。比如，在刚过去的 2020 年，大家在电商平台里准点抢购口罩，上百万人同时在线抢购同一商品，此时就带来了超大并发量。
2. 其次，**秒杀对于扣减的准确性要求极高**。秒杀在绝大部分场景里是一种营销手段，如一元抢 iPhone。商家对有限的商品设置一个亏本价，吸引用户下载或注册 App，达到拉新、提升知名度等目的。因为是亏本营销，如果出现了大面积的超卖，业务上是绝不允许的。

除了秒杀之外，其余的扣减场景，如账户金额的扣减、收费文章免费试读次数的扣减等场景，均很难同时满足上述两个要求，所以它们不是热点扣减的代表性场景。

在如何保障不超卖的问题上，可以直接复用“第 13 讲”和“第 14 讲”的方案。下面将讲解如何应对热点扣减的典型场景“秒杀”带来的百万热点这一挑战。

技术挑战

因为需要保障高可靠的扣减，在应对秒杀时，可以在“第 14 讲”的方案基础上进行升级改造。结合“第 06 讲”介绍的关于热点查询的分析内容，在面对热点扣减时，整个架构图和对应的存储命中如下图 1 所示：

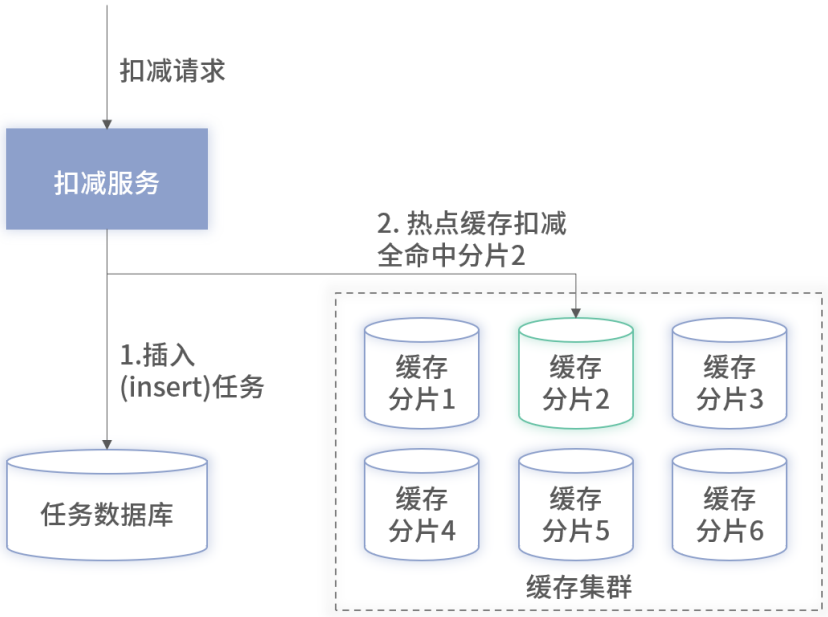

图 1：基于数据库+缓存的热点扣减现状

可以看到，秒杀与热点扣减所带来技术问题是一样的——所有的热点请求均命中同一个存储分片。那为什么不能直接复用“第 06 讲”介绍的“通过增加缓存副本以及使用本地缓存”的方式来应对呢？

下面我们来具体分析一下这其中的原因。

首先，扣减是写请求，即每一次请求都会修改当前商品对应的总数量，且当商品数量扣减为零或当前剩余商品数量小于当次要扣减的数量时，均会返还失败。**而“第 06 讲”热点查询里的缓存副本或者本地缓存里的商品数量均是原始分片的数据镜像**，不能被拿来进行扣减的，否则就会出现数据错乱，甚至超卖的现象。对应的架构示图如下图 2 所示：

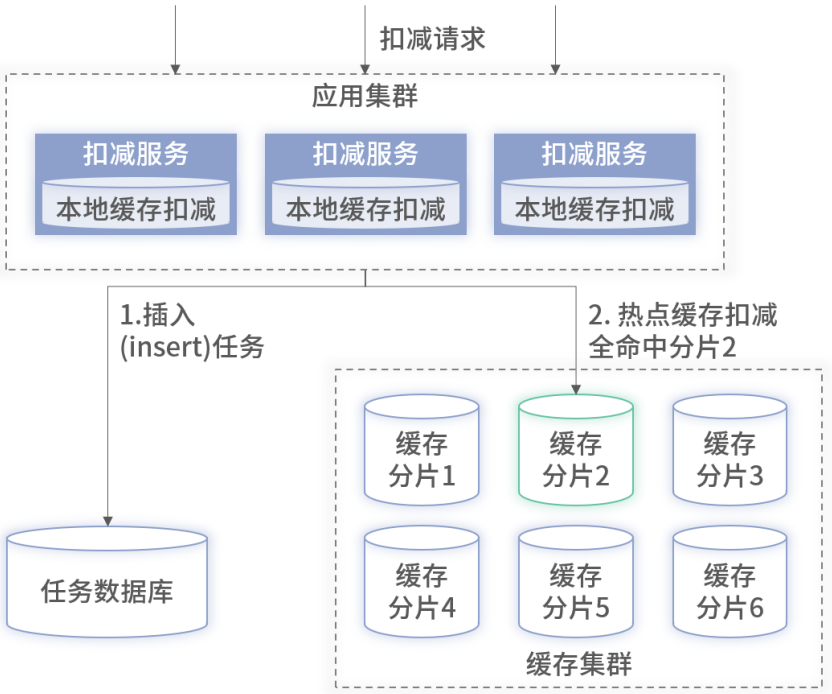

图 2：副本的镜像架构图

其次，本地缓存里的数据是非持久化数据，易丢失。即使将本地缓存持久化至宿主机的磁盘，也会因磁盘故障、不满足 ACID 等原因而导致数据丢失。

如何应对秒杀流量?

既然不能采用热点查询里的方案，只能使用缓存单分片来应对秒杀的流量，但单分片能够支持的流量是有上限。当流量超过上限后如何处理呢?

可以从秒杀的业务上进行分析，你会发现虽然秒杀带来的热点扣减请求非常大，但每次参与秒杀的商品数量是有限的，可能就几百个或者上千个，而热点扣减的流量可能达到上百万。通过简单地计算可以得出，秒杀到商品的概率只有 0.1%，其中 99% 的扣减请求都是“陪跑”的。

这些“陪跑”的请求对于使用者来说可能只是一次简单的点击，但很可能会把正在运行的扣减服务打挂。此时，我们可以对这些瞬间量非常大的“陪跑”请求进行一些前置处理，降低“陪跑”请求的瞬间请求量，或者降低它们对于系统的冲击，此方式就叫作流**量削峰**。体现在流量监控上如下图 3 所示：

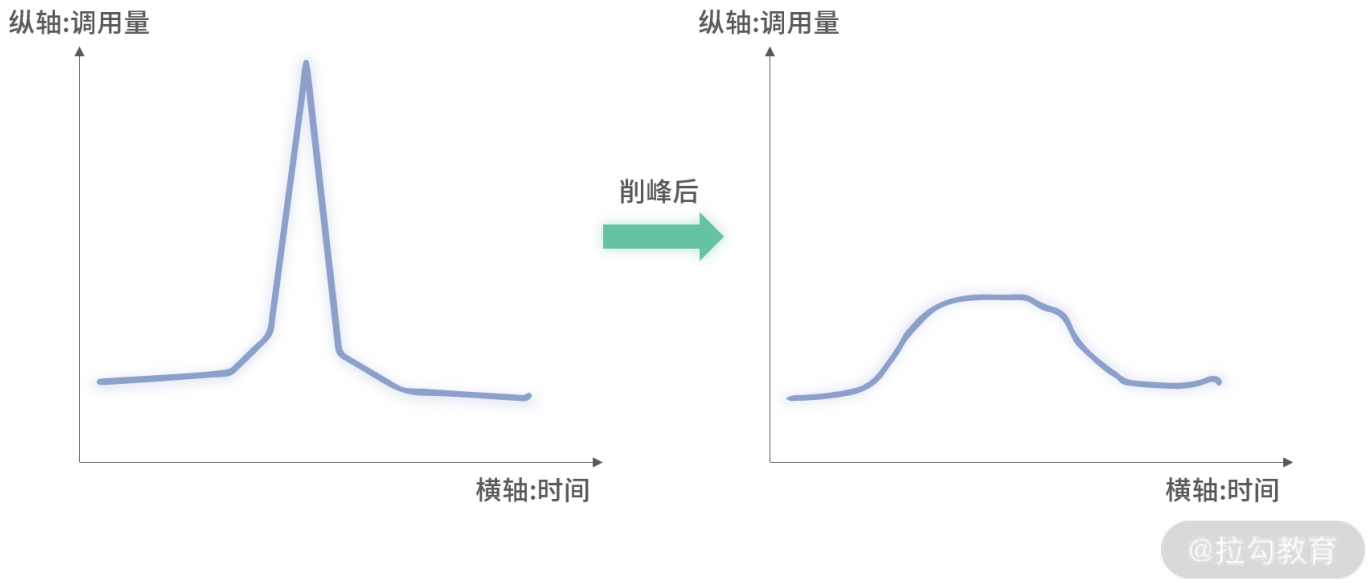

图 3：削峰架构对比图

下面咱们一起讨论下如何进行流量削峰。

如何快速实现流量削峰?

第一步进行的削峰是，先做恶意用户拦截。 上述描述提到过，秒杀通常是基于低价商品的营销活动，抢到商品后转售会有很大的盈利空间。因此，秒杀会吸引来大批的黄牛和黑产用户，对于这些恶意用户可以基于以下几种方式进行拦截：

- **基于用户维护设置限制**。比如同一个账号在 5 秒内最多可以请求扣减多少次。超过该次数，便进行拦截，直接返回失败信息给到商品页面，显示暂时无货。通过此类方式，可以拦截黑产跳过系统界面，直接调用对外暴露的 HTTP 形式的扣减接口所产生的瞬间爆点流量。
- **基于来源 IP 设置限制**。有些黄牛会提前预申请很多账号，因此使用上述账户限制的方式并不能完全拦截住。在账户的基础上，可以对用户的来源 IP 设置限制。比如 5 秒内，同一个 IP 最多可以请求扣减多少次。
- 除了上述方式外，还有很多其他方式可以识别用户，比如现在每一个手机以及电脑都有唯一编码，如手机的 IMEI、电脑的网卡地址等。可以在限制账号、IP 之外，再增加对这些维度的限制。

上述提到的拦截在实现上，可以采用比较成熟的漏桶算法、令牌桶算法。这两个算法在网络上有很多介绍，这里不再赘述。此外，现在有很多开源工具包提供了这两个算法的实现，比如 Java 里的 Guava 包就提供了开箱即用的实现。采用限流算法的架构如下图 4 所示：

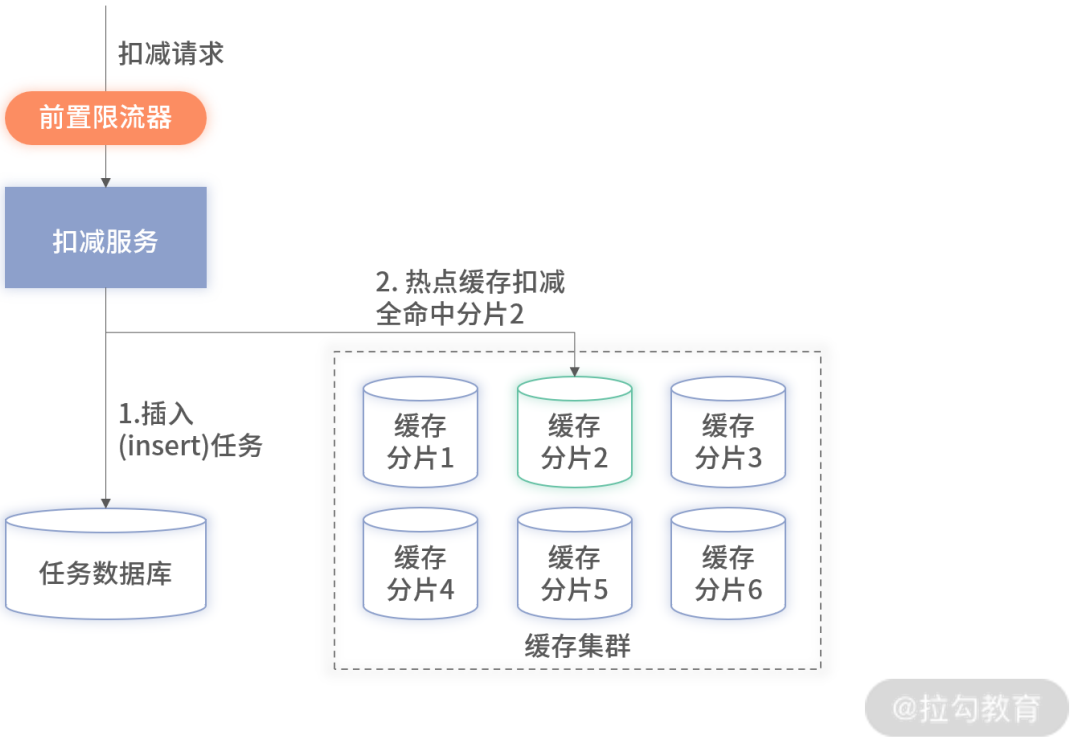

图 4：基于限流的架构图

限流在实现上有两种方式，一种是集中式，另外一种是单机式。集中式是指设置一个总的限流阈值，并将此值存储在一个单独的限流应用中。所有的扣减应用在接收到请求后，均采用远程请求此限流应用的方式，来判断当前是否达到限流值。它的架构如下图 5 所示：

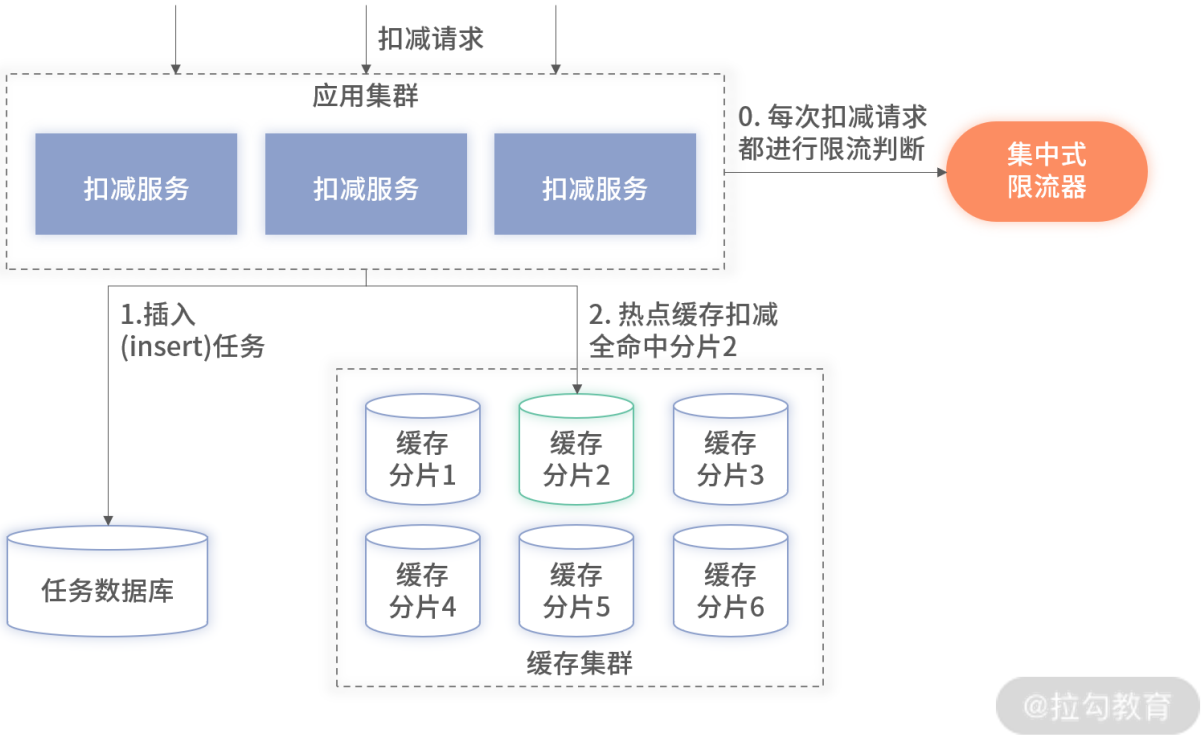

图 5：集中式限流架构

集中式的限流方式最大的好处是**设置简单**，当对整个扣减应用的集群进行极限压测后，得到了极限值。便可以基于此值，设置集群的限流阈值。但这种限流方式也带来了一些问题：

- 首先，调用远程限流服务会增加一次网络消耗，这也降低了扣减服务的性能；

- 其次，远程限流服务提供的限流功能并不精确，因为调用远程的扣减服务会消耗一定的时间，在这个时间区间里，可能会有大批量的热点并发涌入扣减应用，瞬间就会击垮扣减服务；
- 最后，如果所有的请求都要经过限流服务，如何保障限流服务高可用以及能够高效应对热点也是一个难点。

单机式限流是指将上述提到的限流阈值在管理端配置后，主动下发到每一台扣减应用中去，它的架构如下图 6 所示：

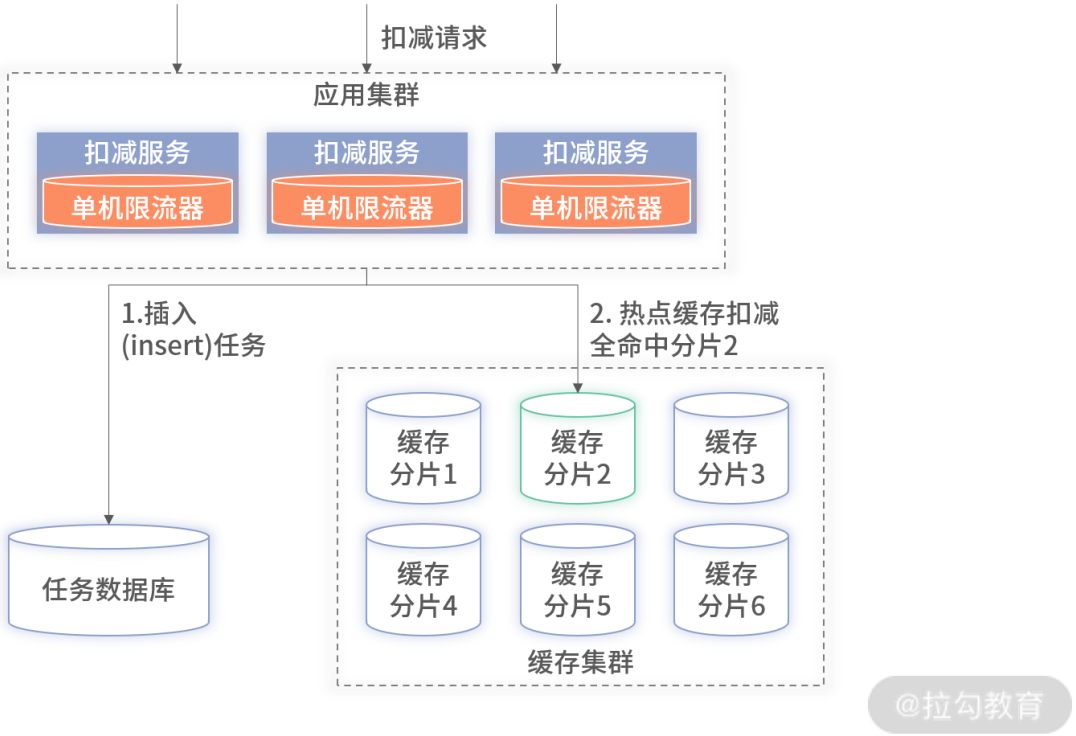

图 6：单机式限流架构

单机式限流是将限流器内置到扣减应用内，可以规避上述集中式限流出现的问题，但它也会带来其他问题：

- 首先，每台机器的限流值需要根据机器的数量实时计算，并将计算后的限流值下发到每台应用机器里，同时更新扣减应用内的限流器；
- 其次，对于扩容的机器需要初始化对应的单机限流器。

在实际的应用中，推荐采用单机维度的限流器，因为它会更加精准和实时。

第二步进行的削峰是，业务层面需要设置权重等级。 秒杀是一种营销活动，营销是有目的的，比如激活许久未下单用户，或者优先让会员抢到商品，增加会员的续费意愿等。

在秒杀接口实现时，可以根据业务规则配置相对应的优先级过滤一些低等级的用户。比如设置高与低的优先级比例为 10：5，它表示在一个时间区间内（如 5 秒），处理 10 个高优先级（如会员用户）的扣减请求时，最多才能处理 5 个低优先级的请求。在实现上，可以使用令牌桶算法，高低优先级各配置一个令牌桶，高优先级的令牌桶数量为 10，低优先级的设置为 5 即可。

第三步进行的削峰是，增加一定的过滤比例。 如果上述两个方式过滤后，热点扣减的并发量仍然较大。可以设置一个固定比例，如 10% 的请求前置过滤并直接返回失败消息，告知用户“抢购火爆，请稍后再试”，也可以降低一部分无效请求。

过滤比例可以根据预估流量和秒杀商品的库存进行设置，如预估流量 50W/S、实际商品库存只有 10 个，那么抢到商品的概率只有 0.002%，抢不到的概率为 99.998%，只要设置过滤率小于抢不到的概率即可。

第四步进行的削峰是，兜底降级不可少。 即使做了上述的限流措施后，流量仍有可能超过“第 14 讲”方案里的单分片的承载最大值，此时，可以从技术层面上增加限流阈值。

首先对缓存的单分片进行压测，得到单分片能够承载的最大值，这个最大值乘以 50% 或者 60% 即可得到缓存单分片线上能够实际承载的最大流量值。之所以要乘以一定比例获得实际承载最大值，是因为在压测时，被压测的缓存单分片的各项指标（如 CPU、网络等）均已达到极限值，系统处在宕机的边缘了。为了保证系统稳定，线上环境的限流值不能设置为此极限值，只能进行一定的折扣。有了单分片的最大承载值，才可以做最后一步的兜底，兜底架构如下图 7 所示：

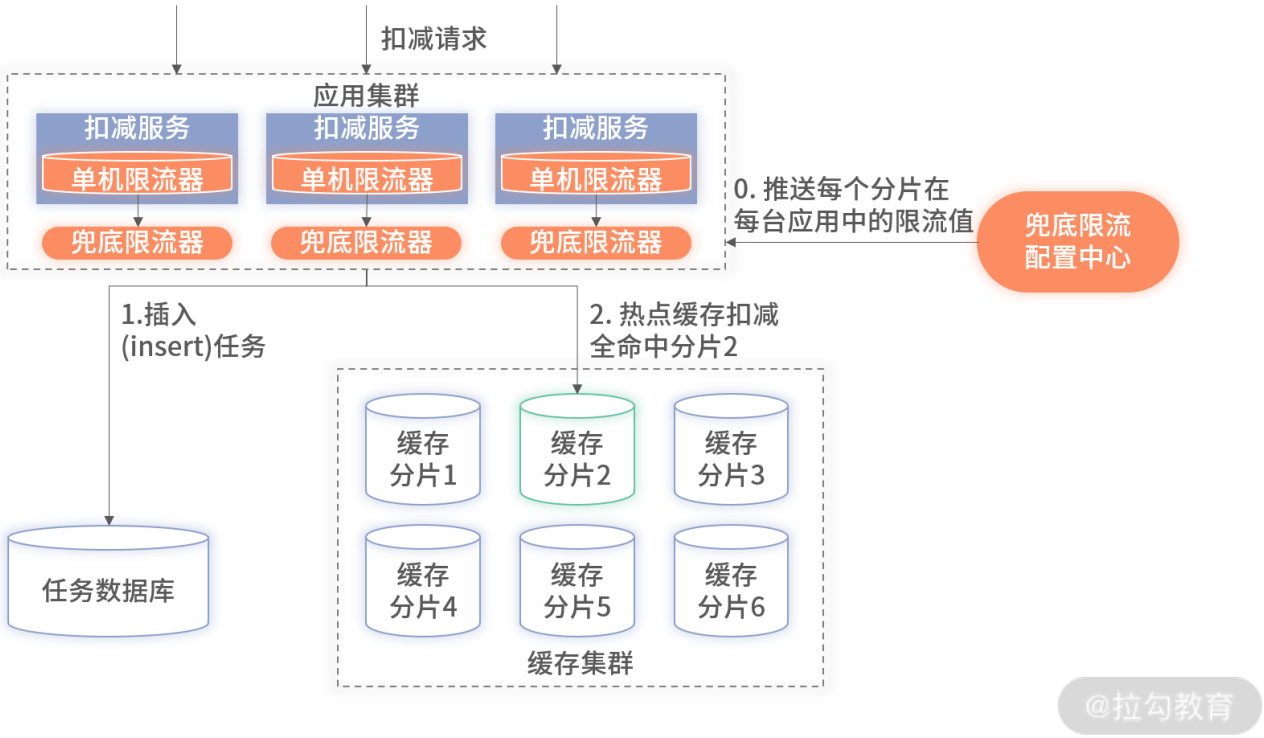

图 7：兜底架构图

在部署的所有扣减应用里，通过上图中编号为 0 的配置中心推送每台机器需要负责的每个缓存分片的限流值（单分片最大承载值/扣减应用机器数），在扣减应用中，按上述推送值，给每一个缓存分片设置一个限流器。

此方案需要扣减应用和缓存中间件有一定的耦合性，即扣减应用需要判断当前请求隶属于哪一个缓存分片。实现上，具体隶属于哪个缓存分片，可以通过缓存中间件提供的路由算法工具来计算。获取到分片标识号后，就可以获取到此标识对应的限流器，然后再进行限流即可。

通过上述方式，即使出现流量超预期，兜底策略既保障了秒杀业务可正常运行，同时又保障了系统不会被打挂。

最后进行的削峰是，售完的商品需前置拦截。 秒杀商品会在瞬间售完，后续所有的请求都会返回无货。对于已经无货的商品，可以采用“第 06 讲”里的方案，将商品已经无货的标记记录在本地缓存里。在秒杀扣减前，先在本地缓存进行判断，如果无货直接返回即可。架构如下图 8 所示：

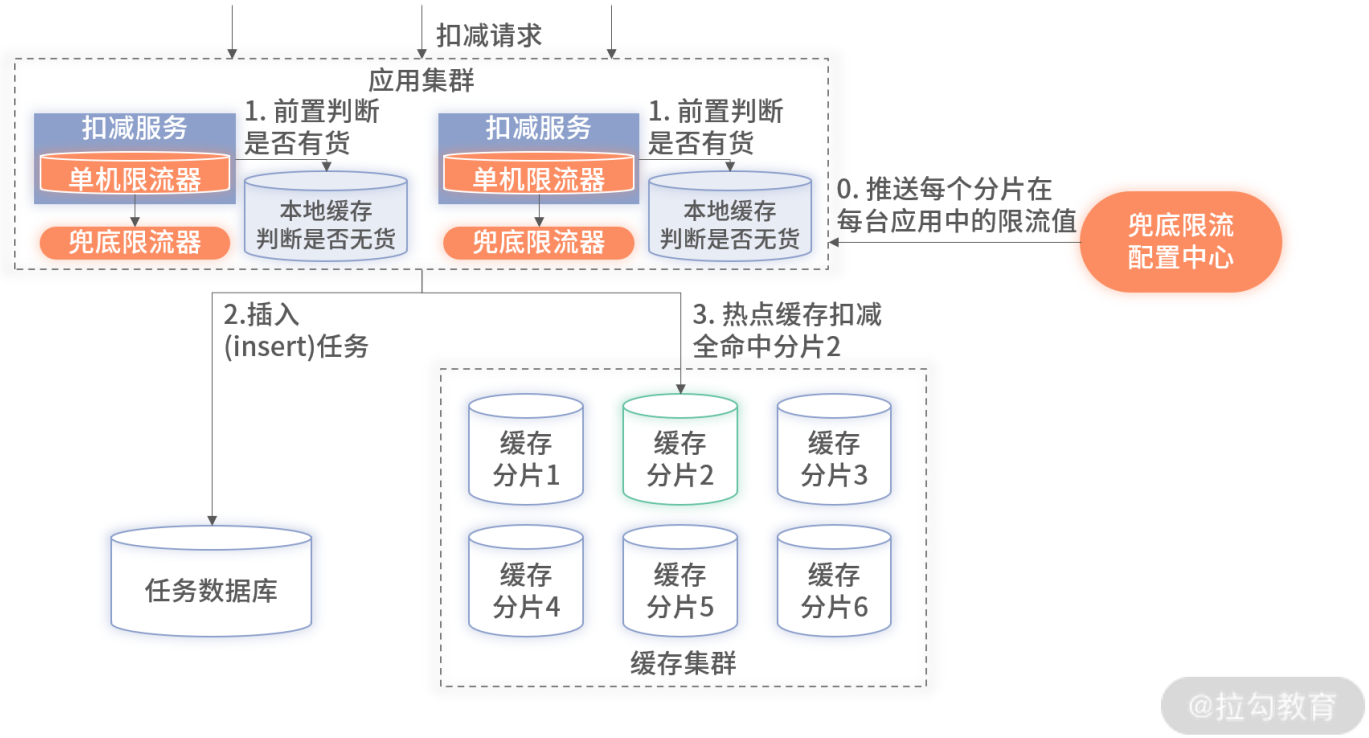

图 8：无货前置拦截

水平扩展架构升级

通过上述几种限流的组合，便可以应对秒杀的热点流量了。但上述的方式会牺牲一定的用户体验，比如按一定比例过滤用户请求、按缓存分片维度过滤用户请求等。

我们可以在上述方案的基础上，做一定的升级来减少有损体验。升级后的架构如下图 9 所示：

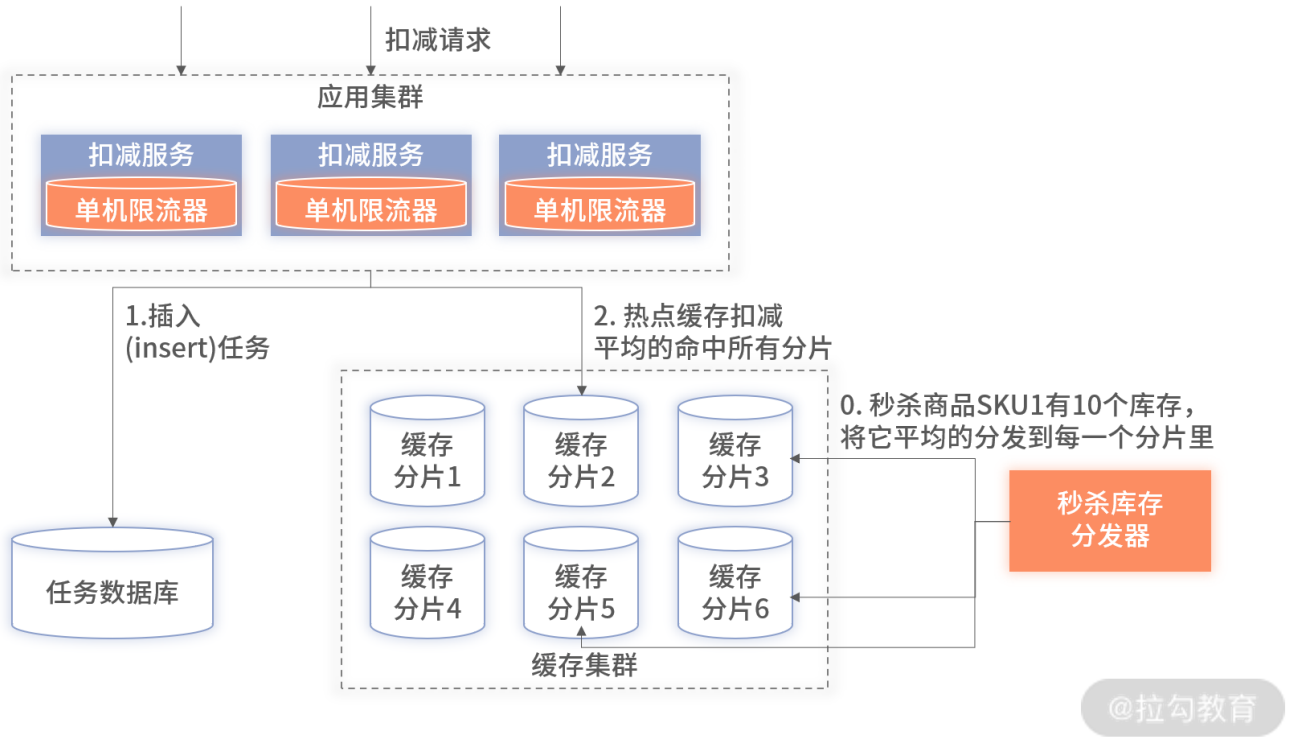

图 9：具备水平扩展的架构

上述架构里，在设置秒杀库存时，将秒杀库存按缓存分片的数量进行平均等分，每一个缓存里均存储一等份即可。比如某一个商品（记为 SKU1）的秒杀库存为 10，当前部署的缓存分片共计 10 个，那么每一个分片里存储该 SKU 的库存数可以为 1，存储在各个缓存里的 key 可以为：SKU1_1、SKU1_2、...、SKU1_10。

在处理秒杀请求时，不只是固定地命中某一个缓存分片，而是在每次请求时轮询命中缓存集群中的每一个缓存分片。

将秒杀商品的库存前置散列到各个缓存分片，可以将原先热点扣减只能使用一个缓存分片升级至多个，提升吞吐量。但此方式有一个弊端，就是更加的定制化。

其他手段

除了上述介绍的手段之外，还有几个方式可以应用在秒杀场景里。

首先，前端静态资源前置。 在秒杀开始之前及在秒杀中，焦急的用户会不断地刷新页面，判断秒杀是否开始，避免自己错过开始时间。刷新秒杀页面其实是热点查询的功能，可以借鉴“第 06 讲”的方式采用应用内的前置缓存解决。对于前台页面上涉及的静态数据，如 JS、CSS、图片等，可以使用 CDN 来提升性能，具体架构如下图 10 所示：

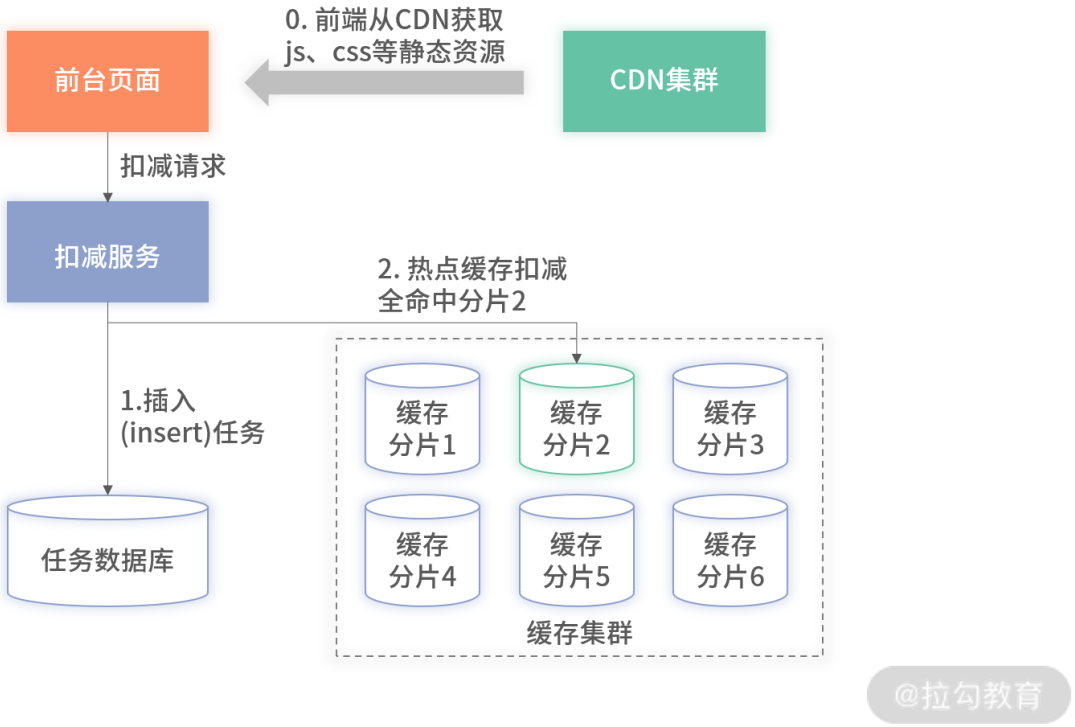

图 10：前端缓存优化架构

其次，业务上隔离。秒杀与正常的购物是有区别的，它是短时间内抢购某一商品。在应对策略上，可以从根据其业务特点进行定制，降低系统的压力。正常的网上购物流程是用户先选购多个商品，加入购物车后再提交订单并进行库存扣减。对于秒杀，可以定制它的前台页面，开发单独的秒杀页面。秒杀开始后，跳过添加购物车的过程，直接提交订单。这样设计，有几个好处。

1. 跳过购物车再提单，增加了用户抢购到商品的概率，提升了用户体验。
2. 业务流程跳过购物车，也降低了热点并发对于购物车后台系统的压力，提升了整体后台系统的稳定性。
3. 秒杀商品直接提单时，就只会有秒杀这一个商品，这对于扣减应用的稳定性有极大的保障。一次扣减只有一个商品相比一次扣减有十几个商品，它在性能、网络带宽的消耗、对于扣减服务的资源占用（如 CPU、内存）等都有更大的节约。

最后，部署隔离。 在完成上述业务隔离后，可以在机器部署上，更往前一步。对于秒杀所涉及的后端应用模块、存储均进行单独部署隔离。通过此种方式，可以更好地应对秒杀，此外也能够减少秒杀的热点并发流量对于原有扣减模块的影响。单独部署的架构如下图 11 所示：

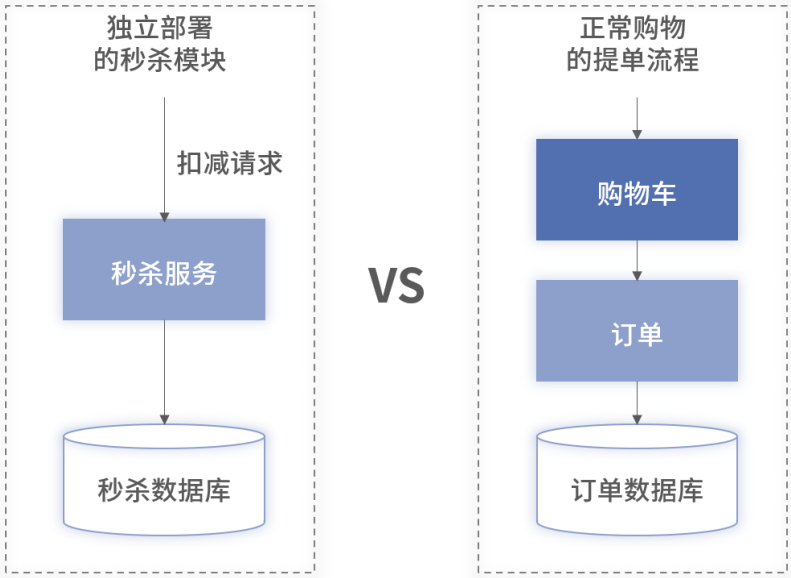

图 11：单独部署的秒杀部署（增加对比正常的扣减）

总结

本讲在本模块前几讲的基础上，介绍了四种限流拦截策略，以及除了限流之外，可以实现水平扩展架构升级的方案。

除了上述方案外，还可以在部署架构、系统隔离、前端静态资源前置等方面进行升级改造来应对热点扣减。

最后，留一道观察题，**你在秒杀抢购商品时，是否遇到过提示你已经无货，后续稍等几秒又抢到的场景呢**？可以参考本讲的内容，思考下背后的原因。

这一讲就到这里，感谢你学习本次课程，接下来我们将学习17 | 如何设计一锤子买卖的 SDK？再见。