B站评论系统架构设计

原创 黄振 哔哩哔哩技术 2022-12-09 12:00 发表于上海

收录于合集
#后端 2 #系统设计 1

本期作者

黄振
社区业务技术组高级开发工程师

01 背景

维基百科对「评论」的定义是：评论是人们对出版物、服务或公司的评估，例如电影（电影评论）、电子游戏（视频游戏评论）、音乐制作（对作品录音作品的音乐评论）、图书（书评）、硬件，如汽车、家用电器或电子计算机、商业软件、活动或表演，例如音乐会、戏剧、音乐剧、舞蹈或艺术展览。除了批判性评论之外，评论的作者还可以对作品进行内容分级以表明其相对价值。

在B站，UP主每天都会发布海量的视频、动态、专栏等内容，随之而来的是弹幕和评论区的各种讨论。播放器中直接滚动播放的弹幕，如同调味剂，重在提升视频观看体验；而点进评论区，相对而言评论文本更长，内容的观点、形式都更丰富，更像是饭后甜点。

随着业务不断发展，B站的评论系统逐渐组件化、平台化；通过持续演进架构设计，管理不断上升的系统复杂度，从而更好地满足各类用户的需求。

02 基础功能模块

评论的基础功能模块是相对稳定的。

1. 发布评论：支持无限盖楼回复。

2. 读取评论：按照时间、热度排序；显示评论数、楼中楼等。

3. 删除评论：用户删除、UP主删除等。

4. 评论互动：点赞、点踩、举报等。

5. 管理评论：置顶、精选、后台运营管理（搜索、删除、审核等）。

结合B站以及其他互联网平台的评论产品特点，评论一般还包括一些更高阶的基础功能：

1. 评论富文本展示：例如表情、@、分享链接、广告等。

2. 评论标签：例如UP主点赞、UP主回复、好友点赞等。

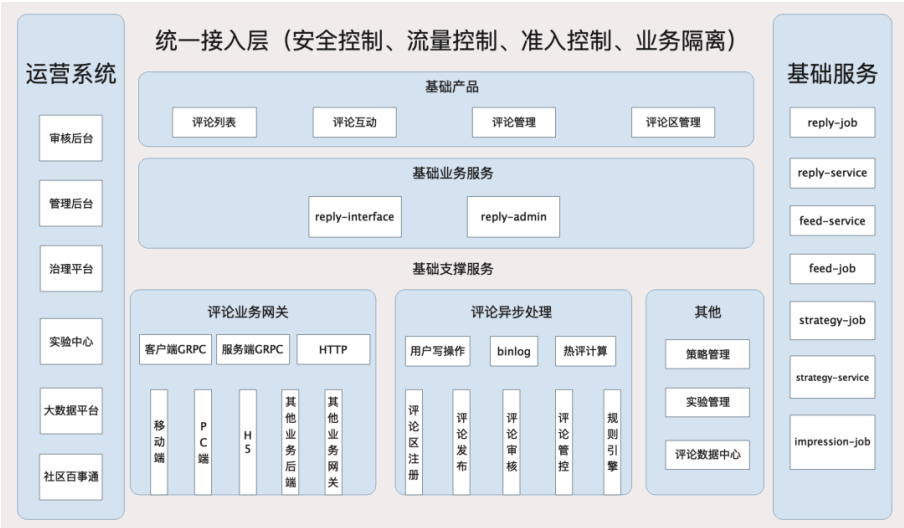
3. 评论装扮：一般用于凸显发评人的身份等。

4. 热评管理：结合AI和人工，为用户营造更好的评论区氛围。

03 架构设计

评论是主体内容的外延。因此一般会作为一个独立系统拆分设计。

3.1 架构设计 - 概览

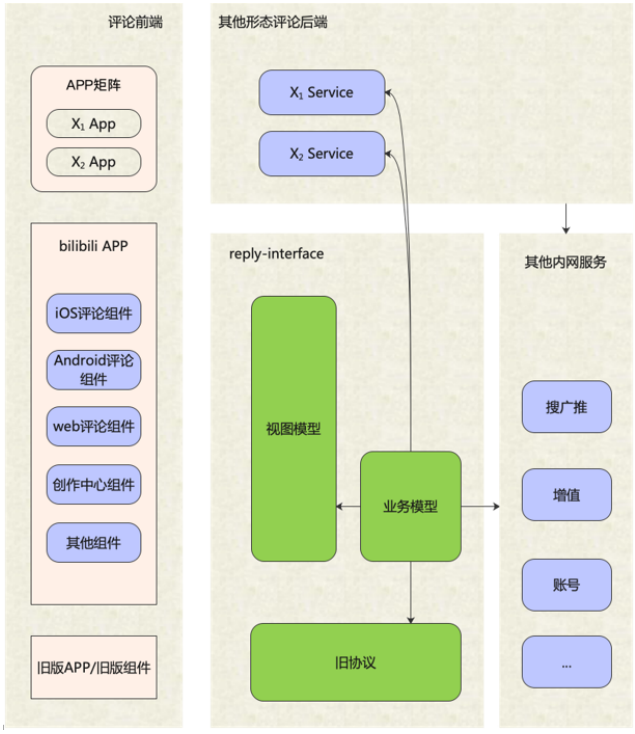


3.2 架构设计 - reply-interface

reply-interface是评论系统的接入层，主要服务于两种调用者：一是客户端的评论组件，二是基于评论系统做二次开发或存在业务关联的其他业务后端。

面向移动端/WEB场景，设计一套基于视图模型的API，利用客户端提供的布局能力，BFF层负责组织业务数据模型，并转换为视图模型，编排后下发给客户端。

面向服务端场景，设计的API需要体现清晰的系统边界，最小可用原则对外提供数据，同时做好安全校验和流量控制。



对评论业务来说，业务数据模型是最为复杂的。B站评论系统历史悠久，承载的功能模块相当之多，其中最核心的是发布类接口以及列表类接口，一写一读，数据字段多、依赖服务多、调用关系复杂，特别是一些依赖的变更，容易造成整个系统的腐化。

因此，我们将整个业务数据模型组装，分为两个步骤，一是服务编排，二是数据组装。服务编排拆分为若干个层级，同一层级的可以并发调用，前置依赖较多的可以流水线调用，结构性提升了复杂调用场景下的接口性能下限；针对不同依赖服务所提供的SLA不同，设置不同的降级处理、超时控制和服务限流方案，保证少数弱依赖抖动甚至完全不可用情况下评论服务可用。数据组装在服务编排之后执行，例如在批量查询评论发布人的粉丝勋章数据之后，将其转换、组装到各个评论卡片之中。

3.3 架构设计 - reply-admin

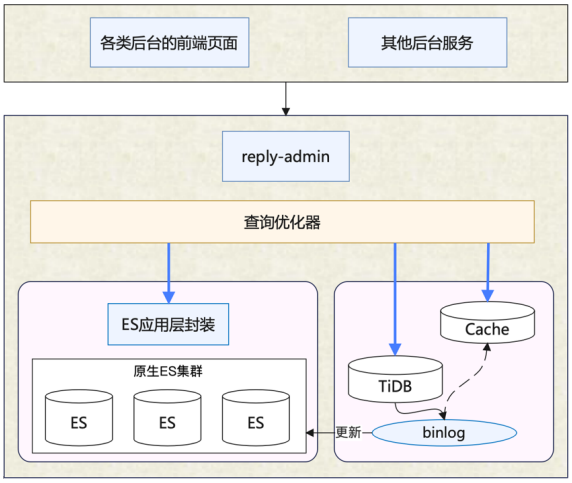
评论管理服务层，为多个内部管理后台提供服务。运营人员的数据查询具有：

1. 组合、关联查询条件复杂；

2. 刚需关键词检索能力；

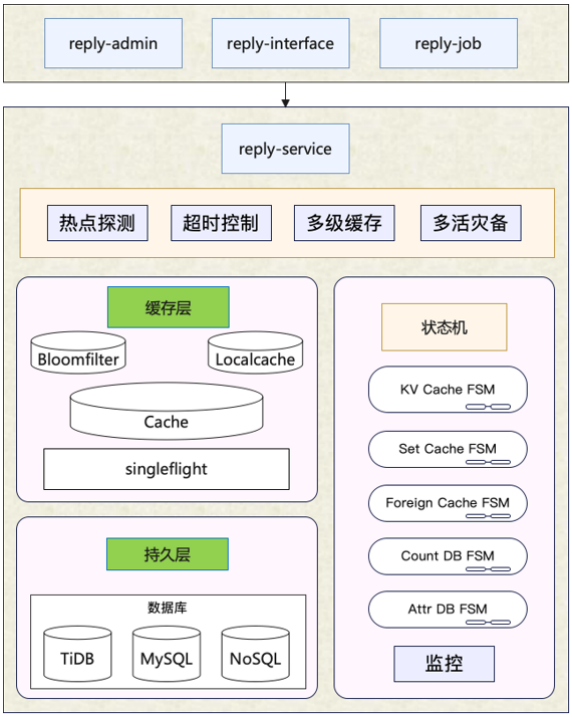
3. 写后读的可靠性与实时性要求高等特征。

此类查询需求，ES几乎是不二选择。但是由于业务数据量较大，需要为多个不同的查询场景建立多种索引分片，且数据更新实时性不高。因此，我们基于ES做了一层封装，提供统一化的数据检索能力，并结合在线数据库刷新部分实时性要求较高的字段。



3.4 架构设计 - reply-service

评论基础服务层，专注于评论功能的原子化实现，例如查询评论列表、删除评论等。一般来说，这一层是较少做业务逻辑变更的，但是需要提供极高的可用性与性能吞吐。因此，reply-service集成了多级缓存、布隆过滤器、热点探测等性能优化手段。

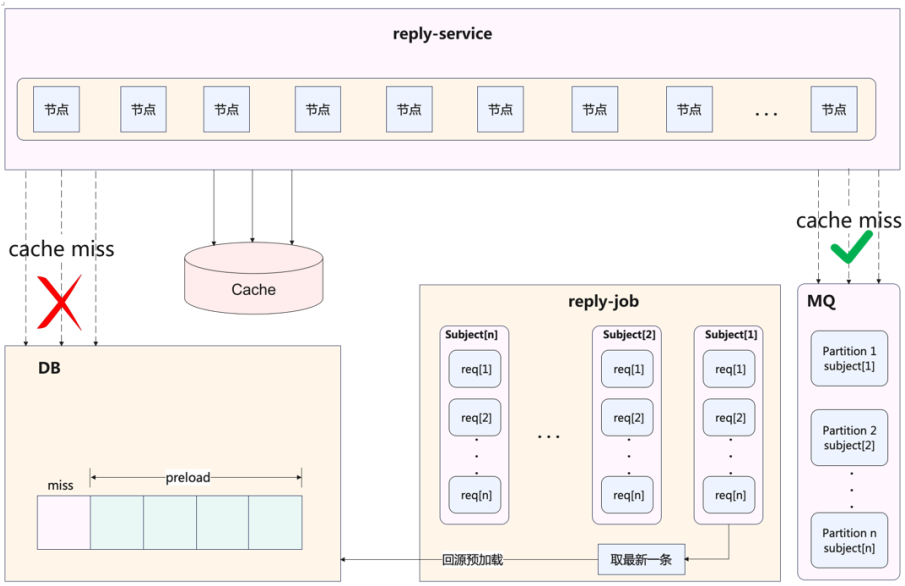


3.5 架构设计 - reply-job

评论异步处理层，主要有两个职责：

1. 与reply-service协同，为评论基础功能的原子化实现，做架构上的补充。

为什么基础功能的原子化实现需要架构的补充呢？最典型的案例就是缓存的更新。一般采用Cache Aside模式，先读缓存，再读DB；缓存的重建，就是读请求未命中缓存穿透到DB，从DB读取到内容之后反写缓存。这一套流程对外提供了一个原子化的数据读取功能。但由于部分缓存数据项的重建代价较高，比如评论列表（由于列表是分页的，重建时会启用预加载），如果短时间内多个服务节点的大量请求缓存未命中，容易造成DB抖动。解决方案是利用消息队列，实现「单个评论列表，只重建一次缓存」。归纳而言，所谓架构上的补充，即是「用单线程解决分布式无状态服务的共性问题」。另一方面，reply-job还作为数据库binlog的消费者，执行缓存的更新操作。



2. 与reply-interface协同，为一些长耗时/高吞吐的调用做异步化/削峰处理。

诸如评论发布等操作，基于安全/策略考量，会有非常重的前置调用逻辑。对于用户来说，这个长耗时几乎是不可接受的。同时，时事热点容易造成发评论的瞬间峰值流量。因此，reply-interface在处理完一些必要校验逻辑之后，会通过消息队列送至reply-job异步处理，包括送审、写DB、发通知等。同时，这里也利用了消息队列的「有序」特性，将单个评论区内的发评串行处理，避免了并行处理导致的一些数据错乱风险。那么异步处理后用户体验是如何保证的呢？首先是C端的发评接口会返回展示新评论所需的数据内容，客户端据此展示新评论，完成一次用户交互。若用户重新刷新页面，因为发评的异步处理端到端延迟基本在2s以内，此时所有数据已准备好，不会影响用户体验。

一个有趣的问题是，早年间评论显示楼层号，楼层号实际是计数器，且在一个评论区范围内不能出现重复。因此，这个楼层发号操作必须是在一个评论区范围内串行的（或者用更复杂的锁实现），否则两条同时发布的评论，获取的楼层号就是重复的。而分布式部署+负载均衡的网关，处理发评论请求是无法实现这种串行的，因此需要放到消息队列中处理。

04 存储设计

4.1 数据库设计

结合评论的产品功能要求，评论需要至少两张表：首先是评论表，主键是评论id，关键索引是评论区id；其次是评论区表，主键是评论区id，平台化之后增加一个评论区type字段，与评论区id组成一个"联合主键"。

由于评论内容是大字段，且相对独立、很少修改，因此独立设计第3张表。主键也是评论id。

评论表和评论区表的字段主要包括4种：

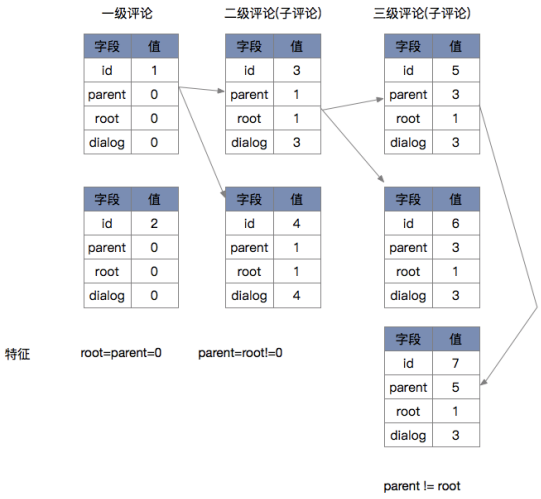
1. 关系类，包括发布人、父评论等，这些关系型数据是发布时已经确定的，基本不会修改。

2. 计数类，包括总评论数、根评论数、子评论数等，一般会在有评论发布或者删除时修改。

3. 状态类，包括评论/评论区状态、评论/评论区属性等，评论/评论区状态是一个枚举值，描述的是正常、审核、删除等可见性状态；评论/评论区属性是一个整型的bitmap，可用于描述评论/评论区的一些关键属性，例如UP主点赞等。

4. 其他，包括meta等，可用于存储一些关键的附属信息。

评论回复的树形关系，如下图所示：



以评论列表的访问为例，我们的查询SQL可能是（已简化）：

1. 查询评论区基础信息：SELECT * FROM subject WHERE obj_id=? AND obj_type=?

2. 查询时间序一级评论列表：SELECT id FROM reply_index WHERE obj_id=? AND obj_type=? AND root=0 AND state=0 ORDER BY floor=? LIMIT 0,20

3. 批量查询根评论基础信息：SELECT * FROM reply_index,reply_content WHERE rpid in (?,?,...)

4. 并发查询楼中楼评论列表：SELECT id FROM reply_index WHERE obj_id=? AND obj_type=? AND root=? ORDER BY like_count LIMIT 0,3

5. 批量查询楼中楼评论基础信息：SELECT * FROM reply_index,reply_content WHERE rpid in (?,?,...)

产品形态上，单个页面只有二级列表（更多嵌套层次，对应嵌套多次点击），且评论计数也只有两级。若回复数也要无限套娃，则一条子评论的发布，需要级联更新所有的父评论的回复数，当前的数据库设计不能满足该需求。

再者，产品侧定义是，若一级评论被删除，其回复也等价于全部删除，若直接删除，此时也可能出现写放大。因此结合查询逻辑，可以不对回复做更新操作，但是评论区的计数更新操作，需要多减去该一级评论的回复数。

评论系统对数据库的选型要求，有两个基本且重要的特征：

1. 必须有事务；

2. 必须容量大。

一开始，我们采用的是MySQL分表来满足这两个需求。但随着B站社区破圈起量，原来的MySQL分表架构很快到达存储瓶颈。于是从2020年起，我们逐步迁移到TiDB，从而具备了水平扩容能力。

4.2 缓存设计

我们基于数据库设计进行缓存设计，选用redis作为主力缓存。主要有3项缓存：

1. subject，对应于「查询评论区基础信息」，redis string类型，value使用JSON序列化方式存入。

2. reply_index，对应于「查询xxx评论列表」，redis sorted set类型。member是评论id，score对应于ORDER BY的字段，如floor、like_count等。

3. reply_content，对应于「查询xxx评论基础信息」，存储内容包括同一个评论id对应的reply_index表和reply_content表的两部分字段。

reply_index是一个sorted set，为了保证数据完整性，必须要判定key存在才能增量追加。由于存在性判定和增量追加不是原子化的，判定存在后、增量追加前可能出现缓存过期，因此选用redis的EXPIRE命令来执行存在性判定，避免此类极端情况导致的数据缺失。此外，缓存的一致性依赖binlog刷新，主要有几个关键细节：

1. binlog投递到消息队列，分片key选择的是评论区，保证单个评论区和单个评论的更新操作是串行的，消费者顺序执行，保证对同一个member的zadd和zrem操作不会顺序错乱。

2. 数据库更新后，程序主动写缓存和binlog刷缓存，都采用删除缓存而非直接更新的方式，避免并发写操作时，特别是诸如binlog延迟、网络抖动等异常场景下的数据错乱。那大量写操作后读操作缓存命中率低的问题如何解决呢？此时可以利用singleflight进行控制，防止缓存击穿。

05 可用性设计

5.1 写热点与读热点

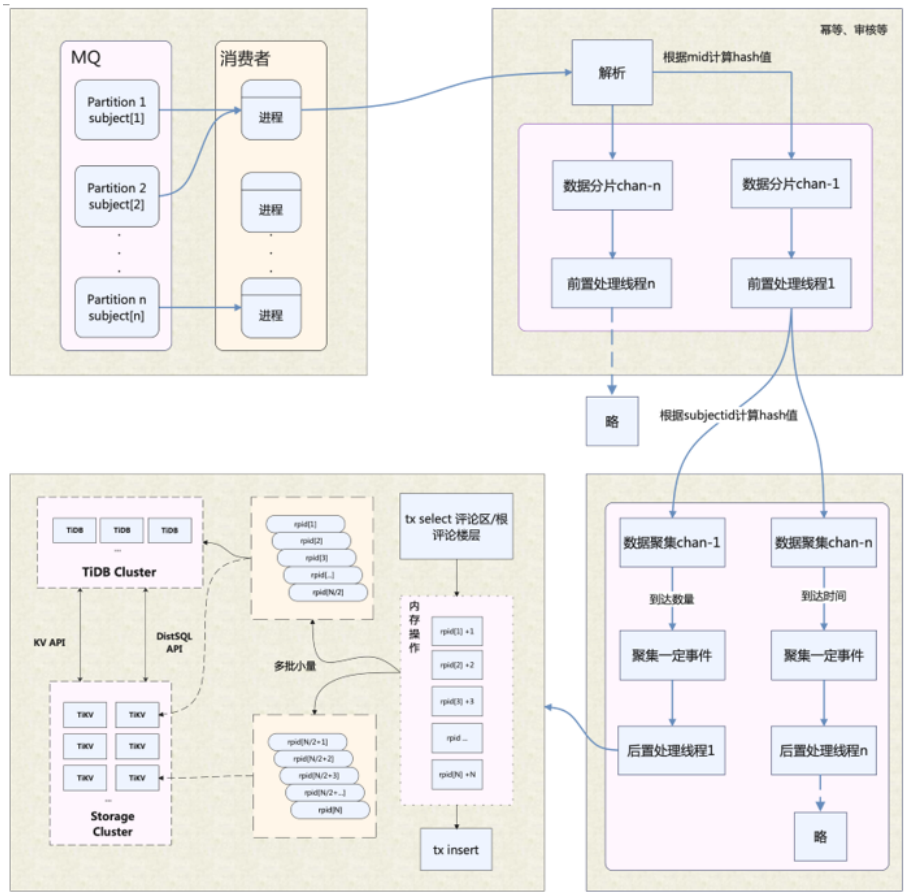
2020年的**腾讯的辣椒酱不香了**[1]，引发一场评论区的狂欢。由于上文所述各类「评论区维度的串行」，当时评论发布的吞吐较低，面对如此大的流量出现了严重延迟。

痛定思痛，我们剖析瓶颈并做了如下优化：

1. 评论区评论计数的更新，先做内存合并再更新，可以减少热点场景下的SQL执行条数；评论表的插入，改成批量写入。

2. 非数据库写操作的其他业务逻辑，拆分为前置和后置两部分，从数据写入主线程中剥离，交由其他的线程池并发执行。

改造后，系统的并发处理能力有了极大提升，同时支持配置并行度/聚合粒度，在吞吐方面具备更大的弹性，热点评论区发评论的TPS提升了10倍以上。



除了写热点，评论的读热点也有一些典型的特征：

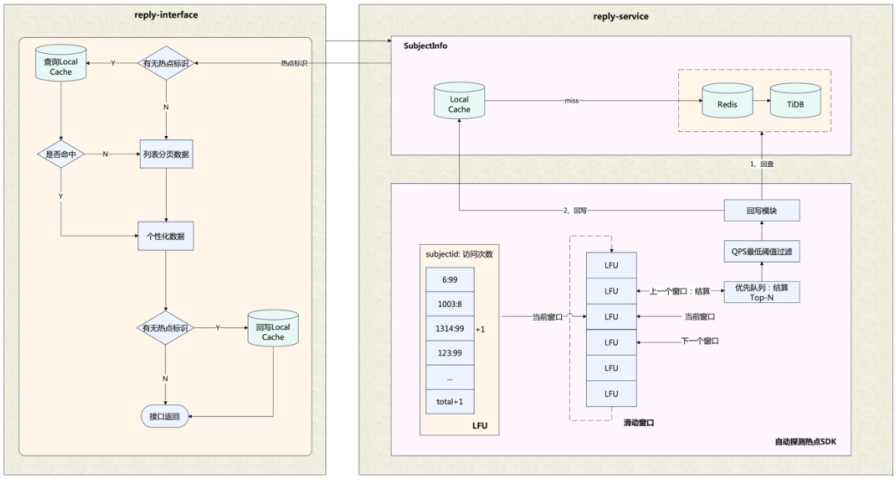
1. 由于大量接口都需要读取评论区基础信息，存在读放大，因此该操作是最先感知到读热点存在的。

2. 由于评论业务的下游依赖较多，且多是批量查询，对下游来说也是读放大。此外，很多依赖是体量相对小的业务单元，数据稀疏，难以承载评论的大流量。

3. 评论的读热点集中在评论列表的第一页，以及热评的热评。

4. 评论列表的业务数据模型也包含部分个性化信息。

因此，我们利用**《直播场景下 高并发的热点处理实践》**[5]一文所使用的SDK，在读取评论区基础信息阶段探测热点，并将热点标识传递至BFF层；BFF层实现了页面请求级的热点本地缓存，感知到热点后即读取本地缓存，然后再加载个性化信息。



热点探测的实现基于单机的滑动窗口+LFU，那么如何定义、计算相应的热点条件阈值呢？

首先，我们进行系统容量设计，列出容量计算的数学公式，主要包括各接口QPS的关系、服务集群总QPS与节点数的关系、接口QPS与CPU/网络吞吐的关系等；然后，收集系统内部以及相应依赖方的一些的热点相关统计信息，通过前面列出的公式，计算出探测数据项

的单机QPS热点阈值。最后通过热点压测，来验证相应的热点配置与代码实现是符合预期的。

5.2 冗余与降级

上文提到，评论基础服务层集成了多级缓存，在上一级缓存未命中或者出现网络错误后，可以视具体场景要求降级至下一级缓存。各级缓存可能有功能上的略微差异，但都能保障用户的基础体验。

此外，评论系统是一个同城读双活的架构。数据库与缓存均是双机房独立部署的，均支持多副本，具备水平扩容的弹性。针对双机房架构下特有的副机房数据延迟故障，支持入口层切流/跨机房重试/应用层补偿，尽可能保证极端情况下用户无感。

在功能层面，我们做了重要级别划分，把相应的依赖划分为强依赖（如审核）、弱依赖（如粉丝勋章）。对于弱依赖，我们一方面在异常情况下坚决限流熔断，另一方面也通过超时控制、请求预过滤、优化调用编排甚至技术方案重构等方式持续优化提升非核心功能的可用性，为业务在评论区获得更好的曝光展现。

06 安全性设计

评论系统的安全性设计可以分为「数据安全」与「舆论安全」。

6.1 数据安全

除了数据安全法所要求的以外，评论系统的数据安全还包括「合规性要求」。评论数据合规，一方面是审核和风控，另一方面对工程侧的要求主要是「状态一致性」。例如，有害评论被删除后，在客户端不能展现，也不能通过API等对外暴露。这就对数据一致性，包括缓存，提出了较高要求。在设计层面主要有两方面实践：

1. 数据读写阶段均考虑了一致性风险，严格保证时序性。

2. 对各类数据写操作，定义了优先级，避免高优先级操作被低优先级操作覆盖，例如审核删除的有害评论，不能被其他普通运营人员/自动化策略放出。

3. 通过冗余校验，避免风险数据外泄。例如评论列表的露出，读取sorted set中的id列表后，还需要校验对应评论的状态，是可见态才允许下发。

6.2 舆论安全

舆论安全问题更为泛化。接口错误导致用户操作失败、关闭评论区、评论计数不准，甚至新功能上线、用户不满意的评论被顶到热评前排等问题均可能引发舆情问题。

在系统设计层面，我们主要通过几方面规避。

1. **不对用户暴露用户无法处理和不值得处理的错误。**例如评论点赞点踩、某个数据项读取失败这一类的轻量级操作，不值得用户重试，此时告知用户操作失败也没有意义。系统可以考虑自行重试，甚至直接忽略。

2. 优化产品功能及其技术实现，例如评论计数、热评排序等。

这里重点介绍一下评论计数的优化思路。

计数不一致的根源，是数据冗余造成的。一般出于性能考虑，会在评论列表以外，给这个列表记录一个长度。也就是用SELECT count FROM subject，代替 SELECT count(*) FROM reply_index。基于这种冗余设计，count字段大部分都是增量更新，即+1、-1，是容易出现误差累积的。

评论计数不准的原因主要有几方面：

1. 并发事务导致的「写倾斜（write skew）」，例如依据评论的状态来做评论区的计数更新。在A事务中读取的评论状态，可能在B事务中被修改，此时A事务计数更新的前提被破坏，也就造成了错误的增量更新。此时计数可能偏大或偏小。

2. 运行时异常、脏数据或者非常规的展示侧控制，导致部分数据被过滤。此时计数可能偏大。

3. 计数冗余同步至其他系统，例如视频表的评论数，延迟导致了过程不一致，同步失败则直接导致最终不一致。

对应并发事务的解决方案，主要有两种：

1. 事务加锁。综合评估而言，对性能的影响较大，特别是存在"锁放大"，越需要加锁的场景，越容易出现"锁冲突"。

2. 串行化。将评论区的所有操作，抽象为一个排队的Domain Events，串行处理，不容易出错。那为什么不能按照评论维度进行拆分，更加不容易出现评论区维度的热点？因为上文提到，删除一级评论时，实际也会从计数上删除其回复；删除二级评论时，也会更新其根评论的计数。多个评论的操作相互影响，因此按照评论维度进行拆分仍然存在并发事务问题。

07 热评设计

7.1 什么是热评

早期的热评，实际就是按照评论点赞数降序。后来衍生了更为复杂的热评：既包括类似「妙评」这种用户推荐、运营精选且带logo突出展示的产品形态，也包括各类热评排序算法，且热评排序算法应用场景也不仅局限于评论主列表的热度序，还包括楼中楼（外露子评论）、动态外露评论等。

热评排序逻辑一般包括点赞数、回复数、内容相关、负反馈数、"时间衰退因子"、字数加权、用户等级加权等等。**如何在B站评论区脱颖而出？**[7]一文从内容运营层面，介绍了什么样的评论更容易上热评前排。

咬文嚼字来说，我们对「热」的理解，大致分为几个阶段：

1. 点赞高，就代表热度高。→ 解决热评的有无问题

2. 基于用户正负样本投票的，加权平均高，就代表热度高。→ 解决高赞高踩的负面热评问题

3. 短时间内点赞率高，就代表热度高。→ 解决高赞永远高赞的马太效应

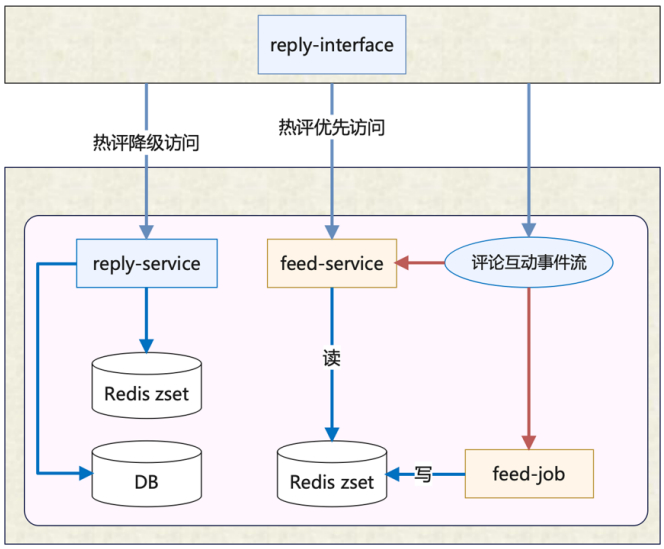
4. 热评用户流量大，社区影响也大。要权衡社会价值观引导、公司战略导向、商业利益、UP主与用户的「情绪」等。→ 追求用户价值平衡

7.2 挑战与应对

显然，我们在不同阶段对热评的理解，在系统设计上也提出了不同层面的要求：

1. 按照点赞绝对值排序，即要实现ORDER BY like_count的分页排序。点赞数是一个频繁更新的值，MySQL，特别是TiDB，由于扫描行数约等于OFFSET，因此在OFFSET较大时查询性能特别差，很难找到一个完美的优化方案。此外，由于like_count的分布可能出现同一个值堆叠多个元素，比如评论区所有的评论都没有赞，我们更多依赖redis的sorted set来执行分页查询，这就要求缓存命中率要非常高。

2. 按照正负样本加权平均的，即**Reddit：威尔逊排序**[6]，到这个阶段，数据库已经无法实现这样复杂的ORDER BY，热评开始几乎完全依赖sorted set这样的数据结构，预先计算好排序分数并写入。于是在架构设计上，新增了feed-service和feed-job来支撑热评列表的读写。



3. 按照点赞率排序，需要实现点赞率的近实时计算。点赞率=点赞数/曝光数，曝光的数据来源是客户端上报的展现日志，量级非常大，可以说是一个写多读少的场景：只有重算排序的时候才会读取曝光数。

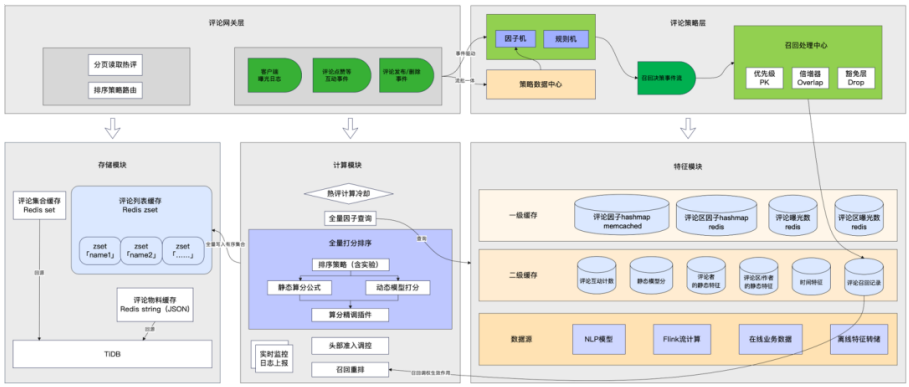
4. 追求用户价值平衡，需要处理各种细分场景下的差异化需求。热评排序与feed排序很像，但也有一点根本性差异：feed排序我们往往都希望是个性化的，每个人看到的都不相同，但评论往往不会如此激进，一般来说会希望大家看到的评论排序都大致相同。由于排序问题的解决方案是探索型的，因此系统设计层面需要提供更多元、更易扩展的工程化能力，包括算法和策略的快速迭代、实验能力等，并提升整个热评模块的可观测水平，监控完善、数据报表丰富、排序过程可解释等等。在架构上，新增了strategy-service和strategy-job来承担这部分策略探索型业务。

此外，数据量级规模的增加，也对系统的吞吐能力提出了更高要求：不管热评的算法如何变化，一般来说，热评列表都需要能够访问到全部评论，且基本维持相同的热评排序逻辑。在评论数过百万甚至千万的评论区，热评排序的挑战点主要在于：

1. 大key问题：例如单个sorted set过大，读写性能都受影响（时间复杂度的基数可以认为都是O(logN)）；全量更新时，还可能遇到redis pipeline的瓶颈。

2. 实时性放大存储压力：多样化的数据源，对特征的导入与更新都提出了挑战，需要支持较丰富的数据结构，和尽可能高的写吞吐（想象一下曝光数作为排序特征的变态要求）；与推荐排序不同，热评排序是全排序，此时需要读取全部评论的全部特征，查询压力也会非常大。

这一阶段，我们仍然在持续优化，在工程落地层面尽可能还原理想的排序算法设计，保障用户的热评浏览体验。目前形成的系统架构总体如下图所示：



图示的「评论策略层」，负责建立一套热评调控体系化能力，通过召回机制来实现想要的“balance”。即先通过策略工程，召回一批应该沉底的不良评论或者应该进前排的优秀评论，然后在排序分计算阶段根据召回结果实现这样的效果。

这样做的好处是，可以**保留一套通用的底层排序算法，然后通过迭代细分场景下的召回策略，来实现差异化评论排序的平衡。**

召回策略的工程设计，按照分层设计的原则拆分为3个部分：

1. 因子机。主要职责是维护策略所需的全部「因子」，包括一些已有的在线/离线数据，也包括为了策略迭代而需要新开发的流式的窗口聚合数据。因子机的重难点是需要管理各种数据获取的拓扑关系，以及通过缓存来保护下游（数据提供方很难也不应该承受热评业务的巨大流量）。所有的因子可以构成一个有向无环图，通过梳理依赖关系和推导计算，实现并发提效、减少冗余。

2. 规则机。实现了一套声明式规则语法，可以直接引用因子机预定义的因子，结合各种逻辑算子构成一个规则表达式。规则机执行命中后，会向下游传递预先声明的召回决策，例如排序提权。

3. 召回处理中心。这一层的职责就是接收规则机返回的各种决策并执行，需要处理不同决策的优先级PK、不同规则的决策叠加作用、决策豁免等。

热评排序涉及的特征，是多数据源的，数据更新方式、更新频率、查询性能也天差万别。因此我们针对数据源的特点做了多级缓存，通过多级冗余与跨级合并，提升了特征读取的稳定性与性能上限。当然，其中的数据实时性、一致性、可用性，仍然处于一个动态权衡取舍的过程。举个例子，曝光数使用redis计数器维护，受限于成本并未持久化；各类静态模型分存在4到5层冗余。此外，还应用了内部稀疏数据的bloom-filter查询、数据局部性集中与散列相结合、近实时大窗口聚合计数等多种性能优化手段。需要指出的是，召回和排序两阶段都需要查询因子/特征，得以复用「因子机」，完成各个特征的差异化实现与维护。

热评排序最关键的计算模块，首先是引入了自适应的冷却算法，根据评论区的评论数、活跃程度等，对重算排序的收益进行预估，拦截了大部分低价值重排请求。其次在全量打分排序阶段，「排序策略」贯穿上下文，既支持传统的静态的经验算分公式，也支持动态的模型打分，支撑AI模型的快速部署快速迭代。通过组合与继承，支持排序策略的叠加、微调，结合评论网关层的排序策略路由，可实现各类定制化排序，完成热评排序系统的平台化。

除了大家点开评论区看到的"热评"，在楼中楼、动态外露评论、评论详情页等类似场景，我们同样应用了这套热评系统，在工程上实现了架构的统一。

■ 7.3 愿景与规划

评论区作为B站社区的重要组成部分，致力于为中文互联网提供一个和谐、有趣的交流环境。另一方面，B站评论区流量巨大，所具备的商业化价值也是需要持续探索的。不管是氛围还是商业，都具有非常强的头部效应。因此，热评，尤其是热评的头部，我们会持续优化产品功能，持续探索排序策略，期望能为用户带来更好的体验：用户可以在这里看到自己喜欢的评论内容，有知识、有温度，也可以看到一些多元化的观点；可以炫一下自己的装扮，也可以守护新人UP主成长；可以倾诉自己的故事，也可以发一条友善的评论，更可以来一个段子，逗乐每一个在互联网里冲浪的有缘人。

以上是今天的分享内容，如果你有什么想法或疑问，欢迎大家在留言区与我们互动，如果喜欢本期内容的话，请给我们点个"在看"吧！

参考资料：

[1] https://t.bilibili.com/406920470238773354?spm_id_from=333.999.0.0

[2] 维基百科：评论（https://zh.wikipedia.org/wiki/%E8%A9%95%E8%AB%96）

[3] 百度评论中台的设计与探索（https://juejin.cn/post/7108973163333025805）

[4] 腾讯老干妈大瓜背后，B站竟成为最大赢家（https://36kr.com/p/783469249310599）

[5] 直播场景下 高并发的热点处理实践（https://www.bilibili.com/read/cv15278397?from=search&spm_id_from=333.337.0.0）

[6] Evan Miller: How Not To Sort By Average Rating（https://www.evanmiller.org/how-not-to-sort-by-average-rating.html）

[7] 如何在B站评论区脱颖而出？

[8] 如何对文章下面的评论做排序(2019年版）（https://zhuanlan.zhihu.com/p/57021517）

哔哩哔哩技术
提供B站相关技术的介绍和讲解
96篇原创内容
公众号

哔哩哔哩招聘
生产快乐的地方
19篇原创内容
公众号

收录于合集 #后端 2

下一篇 · 百亿数据个性化推荐：弹幕工程架构演进 >

喜欢此内容的人还喜欢

翻译软件之🐯——复活谷歌翻译
柯布的花园

大胆！李小龙功夫时钟在跨境将进行一波维权
木瓜侵权

最新版本Minima安卓版本注册教程及节点配置
Minima研究所