# 有赞搜索系统的架构演进

有赞coder 2018-08-31

文 | hehua on PaaS

有赞搜索平台是一个面向公司内部各项搜索应用以及部分 NoSQL 存储应用的 PaaS 产品，帮助应用合理高效的支持检索和多维过滤功能，有赞搜索平台目前支持了大大小小一百多个检索业务，服务于近百亿数据。

在为传统的搜索应用提供高级检索和大数据交互能力的同时，有赞搜索平台还需要为其他比如商品管理、订单检索、粉丝筛选等海量数据过滤提供支持，从工程的角度看，如何扩展平台以支持多样的检索需求是一个巨大的挑战。

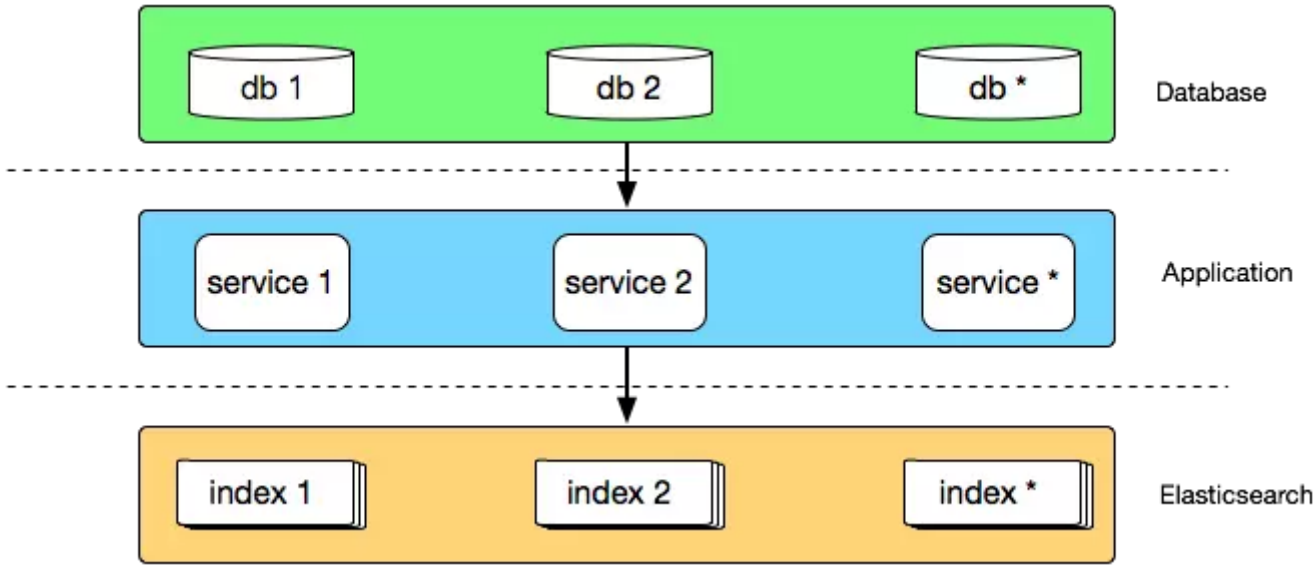
我是有赞搜索团队的第一位员工，也有幸负责设计开发了有赞搜索平台到目前为止的大部分功能特性，我们搜索团队目前主要负责平台的性能、可扩展性和可靠性方面的问题，并尽可能降低平台的运维成本以及业务的开发成本。

## Elasticsearch

Elasticsearch 是一个高可用分布式搜索引擎，一方面技术相对成熟稳定，另一方面社区也比较活跃，因此我们在搭建搜索系统过程中也是选择了 Elasticsearch 作为我们的基础引擎。

## 架构1.0

时间回到 2015 年，彼时运行在生产环境的有赞搜索系统是一个由几台高配虚拟机组成的 Elasticsearch 集群，主要运行商品和粉丝索引，数据通过 Canal 从 DB 同步到 Elasticsearch，大致架构如下：




通过这种方式，在业务量较小时，可以低成本的快速为不同业务索引创建同步应用，适合业务快速发展时期，但相对的每个同步程序都是单体应用，不仅与业务库地址耦合，需要适应业务库快速的变化，如迁库、分库分表等，而且多个 canal 同时订阅同一个库，也会造成数据库性能的下降。

另外 Elasticsearch 集群也没有做物理隔离，有一次促销活动就因为粉丝数据量过于庞大导致 Elasticsearch 进程 heap 内存耗尽而 OOM，使得集群内全部索引都无法正常工作，这给我上了深深的一课。

## 架构 2.0

我们在解决以上问题的过程中，也自然的沉淀出了有赞搜索的 2.0 版架构，大致架构如下：

首先数据总线将数据变更消息同步到 mq，同步应用通过消费 mq 消息来同步业务库数据，借数据总线实现与业务库的解耦，引入数据总线也可以避免多个 canal 监听消费同一张表 binlog 的虚耗。

# 高级搜索（Advanced Search）

随着业务发展，我们也逐渐出现了一些比较中心化的流量入口，比如分销、精选等，这时普通的 bool 查询并不能满足我们对搜索结果的细粒率排序控制需求，将复杂的 function_score 之类专业性较强的高级查询编写和优化工作交给业务开发负责显然是个不可取的选项，这里我们考虑的是通过一个高级查询中间件拦截业务查询请求，在解析出必要的条件后重新组装为高级查询交给引擎执行，大致架构如下：



这里另外做的一点优化是加入了搜索结果缓存，常规的文本检索查询 match 每次执行都需要实时计算，在实际的应用场景中这并不是必须的，用户在一定时间段内（比如 15 或 30 分钟）通过同样的请求访问到同样的搜索结果是完全可以接受的，在中间件做一次结果缓存可以避免重复查询反复执行的虚耗，同时提升中间件响应速度，对高级搜索比较感兴趣的同学可以阅读另外一篇文章《有赞搜索引擎实践（工程篇）》（见技术博客），这里不再细述。

# 大数据集成

搜索应用和大数据密不可分，除了通过日志分析来挖掘用户行为的更多价值之外，离线计算排序综合得分也是优化搜索应用体验不可缺少的一环，在 2.0 阶段我们通过开源的 es-hadoop 组件搭建 hive 与 Elasticsearch 之间的交互通道，大致架构如下：

通过 flume 收集搜索日志存储到 hdfs 供后续分析，也可以在通过 hive 分析后导出作为搜索提示词，当然大数据为搜索业务提供的远不止于此，这里只是简单列举了几项功能。

# 问题

这样的架构支撑了搜索系统一年多的运行，但是也暴露出了许多问题，首当其冲的是越发高昂的维护成本，除去 Elasticsearch 集群维护和索引本身的配置、字段变更，虽然已经通过数据总线与业务库解耦，但是耦合在同步程序中的业务代码依旧为团队带来了极大的维护负担。消息队列虽然一定程序上减轻了我们与业务的耦合，但是带来的消息顺序问题也让不熟悉业务数据状态的我们很难处理。这些问题我总结在之前写过的一篇文章。

除此之外，流经 Elasticsearch 集群的业务流量对我们来说呈半黑盒状态，可以感知，但不可预测，也因此出现过线上集群被内部大流量错误调用压到CPU占满不可服务的故障。

# 目前的架构 3.0

针对 2.0 时代的问题，我们在 3.0 架构中做了一些针对性调整，列举主要的几点：

1. 通过开放接口接收用户调用，与业务代码完全解耦；
2. 增加 proxy 用来对外服务，预处理用户请求并执行必要的流控、缓存等操作；
3. 提供管理平台简化索引变更和集群管理 这样的演变让有赞搜索系统逐渐的平台化，已经初具了一个搜索平台的架构：

## Proxy

作为对外服务的出入口，proxy 除了通过 ESLoader 提供兼容不同版本 Elasticsearch 调用的标准化接口之外，也内嵌了请求校验、缓存、模板查询等功能模块。

请求校验主要是对用户的写入、查询请求进行预处理，如果发现字段不符、类型错误、查询语法错误、疑似慢查询等操作后以 fast fail 的方式拒绝请求或者以较低的流控水平执行，避免无效或低效能操作对整个 Elasticsearch 集群的影响。

缓存和 ESLoader 主要是将原先高级搜索中的通用功能集成进来，使得高级搜索可以专注于搜索自身的查询分析和重写排序功能，更加内聚。我们在缓存上做了一点小小的优化，由于查询结果缓存通常来说带有源文档内容会比较大，为了避免流量高峰频繁访问导致 codis 集群网络拥堵，我们在 proxy 上实现了一个简单的本地缓存，在流量高峰时自动降级。

这里提一下模板查询，在查询结构（DSL）相对固定又比较冗长的情况下，比如商品类目筛选、订单筛选等，可以通过模板查询（search template）来实现，一方面简化业务编排DSL的负担，另一方面还可以通过编辑查询模板 template，利用默认值、可选条件等手段在服务端进行线上查询性能调优。

## 管理平台

为了降低日常的索引增删、字段修改、配置同步上的维护成本，我们基于 Django 实现了最初版本的搜索管理平台，主要提供一套索引变更的审批流以及向不同集群同步索引配置的功能，以可视化的方式实现索引元数据的管理，减少我们在平台日常维护上的时间成本。

由于开源 head 插件在效果展示上的不友好，以及暴露了部分粗暴功能：



如图，可以通过点按字段使得索引按指定字段排序展示结果，在早期版本 Elasticsearch 会通过 fielddata 加载需要排序的字段内容，如果字段数据量比较大，很容易导致 heap 内存占满引发 full gc 甚至 OOM，为了避免重复出现此类问题，我们也提供了定制的可视化查询组件以支持用户浏览数据的需求。

## ESWriter

由于 es-hadoop 仅能通过控制 map-reduce 个数来调整读写流量，实际上 es-hadoop 是以 Elasticsearch 是否拒绝请求来调整自身行为，对线上工作的集群相当不友好。为了解决这种离线读写流量上的不可控，我们在现有的 DataX 基础上开发了一个 ESWriter 插件，能够实现记录条数或者流量大小的秒级控制。

## 挑战

平台化以及配套的文档体系完善降低了用户的接入门槛，随着业务的快速增长，Elasticsearch 集群本身的运维成本也让我们逐渐不堪，虽然有物理隔离的多个集群，但不可避免的会有多个业务索引共享同一个物理集群，在不同业务间各有出入的生产标准上支持不佳，在同一个集群内部署过多的索引也是生产环境稳定运行的一个隐患。

另外集群服务能力的弹性伸缩相对困难，水平扩容一个节点都需要经历机器申请、环境初始化、软件安装等步骤，如果是物理机还需要更长时间的机器采购过程，不能及时响应服务能力的不足。

## 未来的架构 4.0

当前架构通过开放接口接受用户的数据同步需求，虽然实现了与业务解耦，降低了我们团队自身的开发成本，但是相对的用户开发成本也变高了，数据从数据库到索引需要经历从数据总线获取数据、同步应用处理数据、调用搜索平台开放接口写入数据三个步骤，其中从数据总线获取数据与写入搜索平台这两个步骤在多个业务的同步程序中都会被重复开发，造成资源浪费。这里我们目前也准备与 PaaS 团队内自研的DTS（Data Transporter，数据同步服务）进行集成，通过配置化的方式实现 Elasticsearch 与多种数据源之间的自动化数据同步。

要解决共享集群应对不同生产标准应用的问题，我们希望进一步将平台化的搜索服务提升为云化的服务申请机制，配合对业务的等级划分，将核心应用独立部署为相互隔离的物理集群，而非核心应用通过不同的应用模板申请基于 k8s 运行的 Elasticsearch 云服务。应用模板中会定义不同应用场景下的服务配置，从而解决不同应用的生产标准差异问题，而且云服务可以根据应用运行状况及时进行服务的伸缩容。

## 小结

本文从架构上介绍了有赞搜索系统演进产生的背景以及希望解决的问题，涉及具体技术细节的内容我们将会在本系列的下一篇文章中更新。