**[#研发解决方案介绍#基于ES的搜索+筛选+排序解决方案](http://www.cnblogs.com/zhengyun_ustc/p/55solution6.html)**

郑昀 基于胡耀华和王超的设计文档 最后更新于2014/12/3

关键词：ElasticSearch、Lucene、solr、搜索、facet、高可用、可伸缩、mongodb、SearchHub、商品中心

本文档适用人员：研发和运维

提纲：

1. 曾经的基于MongoDB的筛选+排序解决方案
2. MongoDB方案的缺陷
3. 看中了搜索引擎的facet特性
4. 看中了ES的简洁
5. 看中了ES的天生分布式设计
6. 窝窝的ES方案
7. ES的几次事故和教训
8. ES自身存在的问题

  首先要感谢王超和胡耀华两位研发经理以严谨治学的研究精神和孜孜以求的工作态度给我们提供了高可用、可伸缩、高性能的ES方案。

**一，曾经的基于 MongoDB 的筛选+排序解决方案**

  电商的商品展示无非“List（列表页）-Detail（详情页）”模式。生活服务电商更特殊一点，不同开站城市下的用户看到的团购/旅游/酒店/抽奖/电影订座/外卖…等商品集合以及排序也不一样。

  起初窝窝的 List 需求比较简单，所以用 memcached+mysql 也就解决了，但随着在 List 页做多级筛选，根据排序公式计算商品得分来做自动排序等需求的提出，我们把视线转向了 MongoDB。

  2012年，我们针对窝窝当时的 MongoDB 实现方案进一步提出，商品中心的改造思路为“**持久化缓存模式，尽量减少接口调用**”：

|  |
| --- |
| * 商品中心小组**对外提供的实际上是一个存储介质** * 把本需做复杂关联查询的商品数据（base属性集合、ext属性集合、BLOB集合）**组装成一个 Document 放入 MongoDB**等持久化存储介质中 * 允许不同商品具有不同属性的可扩展性 * 商品中心要做的是维护好这个存储介质，保证：   + 商品数据的准确性：     - 如商品自然下线，从介质中清除；     - 如商品紧急下线，默认保留一段时间如6小时；     - 如商品base/ext/blob属性发生变更，有不同的时间策略来更新，如base属性改变，则需要第一时间更新；   + 商品可按常见规则快速抽取：     - 如view层按频道+城市抽取商品，     - 如view层按城市+区县+前台分类抽取商品， * view层可由各个系统自行开发 |

  这样，MongoDB 里不仅仅存储了一份份 documents，还存储了不同开站城市、不同频道、不同排列组合下商品列表的 Goods IDs 清单。排序基本靠 MongoDB 排。ids 清单定时更新。

  这之后，商品中心分拆为：泰山和 GoodsCenter 两部分。

**二，MongoDB 方案的缺陷**

  随着网站业务的不断发展，网站商品搜索筛选的粒度越来越细，维度也就越来越多，多维度的 count 和 select 查询，业务上各种排序需求，使 MongoDB 集群压力山大，以至于屡屡拖累商品中心和泰山的性能。

  2012年下半年，我们意识到：

|  |
| --- |
| 由于频道页流量小于首页，尤其是用户很少点击到的深度筛选条件组合查询，所以下图中的所有枚举项商品数量都容易缓存失效或缓存挤出：  http://images.cnitblog.com/blog/7438/201412/121854573537183.png  图2 筛选越来越复杂，标题数字却要保持准确性    一旦缓存失效后，但凡我从上图的“20元以下”点击切换到“51-80元”或做更深层次筛选，那么程序就要针对上面所有组合条件对 MongoDB 商品记录逐一做 count 计算。    虽然每一个 count 计算都很快不属于慢查询，但也架不住多啊，尤其是配上区县和商圈等动辄6、7层深的筛选组合，点击一次轻易就涉及成百次的 count 计算，代价还是很大的。    由于在商城模式下，不同频道很可能不断增加新筛选条件，**导致筛选组合越来越复杂，最终可能要求我们从基于 NoSQL 的排序和筛选方案，尽快转变为基于搜索引擎的排序和筛选方案**。 |

  2012年时，不同筛选维度的组合筛选造成 **MongoDB 的索引命中率不高**，MongoDB一旦没有命中索引，**其查询效率会直线下降**，从而造成整个MongoDB的压力增大响应变慢（MongoDB 的索引策略基本和 MySQL 的差不多）。有段时间，我们不止一次遇到由于 MongoDB 的慢查，拖挂所有前台工程的情况，焦头烂额。

  商品中心需要升级。**技术选型主要集中在 solr 和 ES 这两个均构建于 Lucene 之上的搜索引擎**。

  这时，我们也注意到了外界对新生事物 Elastic Search 的各种溢美之辞，系统运维部此前也用 Logstash+ElasticSearch+Kibana 方案替代了 Splunk，也算是对 ES 的搭建有了一定了解。

  那时还看到了专门做 solr vs es 的网站：<http://solr-vs-elasticsearch.com/>

**三，看中了搜索引擎的 facet 特性**

  借用腾讯一篇博文来讲解 facet search：(分面搜索，facet：方面，侧面)

|  |
| --- |
| **介绍分面**  　　分面是指事物的多维度属性。例如一本书包含主题、作者、年代等分面。而分面搜索是指通过事物的这些属性不断筛选、过滤搜索结果的方法。可以将分面搜索看成搜索和浏览的结合。  **灵活使用分面** 　　分面不但可以用来筛选结果，也可以用来对结果排序。电商网站中常用风格、品牌等分面筛选搜索结果，而价格、信誉、上架时间等分面则用来排序。  　　有时用户并不明确自己的目的，因此提供宽松的筛选方式更符合这部分用户的预期。Bing 的旅行搜索中选择航班时，用户可以通过滑块来选择某个时间段起飞的航班。 |

  facet 的字段必须被索引，无需分词，无需存储。无需分词是因为该字段的值代表了一个整体概念，无需存储是因为一般而言用户所关心的并不是该字段的具体值，而是作为对查询结果进行分组的一种手段，用户一般会沿着这个分组进一步深入搜索。

**facet 特性对我们最大优点是，查询结果里自带 count 信息，无需我们单独计算不同排列组合的 count 信息，一举扫清性能瓶颈。**

  solr 里 facet search 分为三种类型：

1. Field Facet：如果需要对多个字段进行Facet查询，那么将 facet.field 参数声明多次，Facet字段必须被索引；
2. Date Facet：时间字段的取值有无限性，用户往往关心的不是某个时间点而是某个时间段内的查询统计结果，譬如按月份查；
3. Facet Query：利用类似于filter query的语法提供了更为灵活的Facet，譬如根据价格字段查询时，可设定不同价格区间；

**四，看中了 ES 的简洁**

  2012年下半年不少人倾倒于 ES 的简洁之美：

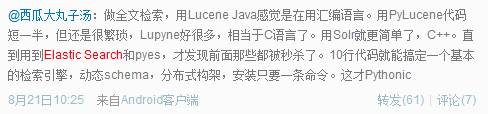


图3

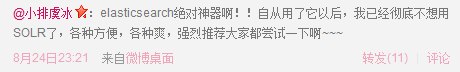


图4

  ES 的优点：

* 简单
* RESTful
* json 格式 Response
* 天生分布式
* [Querydsl](http://www.querydsl.com/) 风格查询

**五，看中了 ES 的天生分布式**

  ES 毕竟是后来者，所以可以说为分布式而生。它的处理能力上，支持横向扩展，理论上无限制；存储能力上，取决于磁盘空间（根据提取字段的数量，索引后的数据量是原始数据量的几倍，譬如我们的 Logstash+ES 方案中对 nginx 访问日志提取了17个字段（都建立了索引），存储数据量8倍于原始日志）。

  比如在高峰期，我们可以采用调配临时节点的方式，来分解压力，在不需要的时候我们可以停掉多余的节点来节省资源。

  还有ES的高可用性，在集群节点出现一个节点或者多个节点出现故障时，主要数据完整，依然可以正常提供服务。

  这里有一个数据，大概是2013年时，有一个[访谈提及 Github 是如何使用 ES](http://exploringelasticsearch.com/github_interview.html)：1，用了 40～50 个索引，包括库、用户、问题、pull请求、代码、用户安全日志、系统异常日志等等；2，44台 EC2 主机处理 30T 的数据，每台机器配备 2T SSD 存储；3，8台机器仅仅用于搜索，不保存数据。当然，Github 也曾经在 ES 升级上栽过大跟头，那是2013年1月17日的事儿了，参考《2013，GitHub使用elasticsearch遇到的一些问题及解决方法，[中文译稿](http://www.searchtech.pro/github-elasticsearch-usecase)，[英文原文](https://github.com/blog/1397-recent-code-search-outages)》。

**六，窝窝的 ES 方案**

**6.1>架出一层 SearchHub**

  所有数据查询均通过 SearchHub 工程完成，如下图所示：

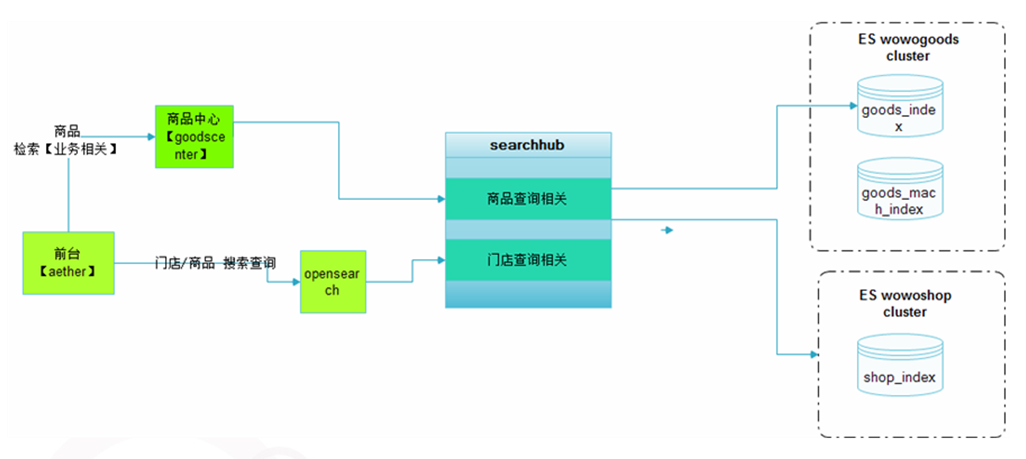


图5 SearchHub

**6.2>通过 NotifyServer 来异步更新各个系统**

  窝窝的数据更新基本都是通过 [notify （基于中间件 NotifyServer）](http://www.cnblogs.com/zhengyun_ustc/p/55solution4.html)的形式来保证各个系统的数据一致性。

**6.3>索引设计方案**

**6.3.1>商品索引设计**

  商品维度是我们主要的查询维度，其业务复杂度也比较高。针对网站查询特性，我们的商品主索引方案为：每个城市建立一个 index，所以一共有400多个 index，每个城市仅有1个主 shard（不分片）。这样做的好处是以后我们根据热点城市和非热点城市，可以将各个 index 手工分配到不同的 node 上，可以做很多优化。

  其结构为：

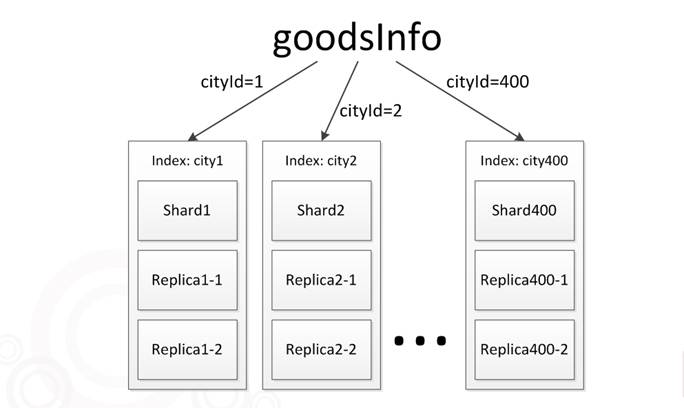


图7 goodsinfo

  为了减少索引量和功能拆分，减少商品索引的内存占用，所以我们把全文检索单独建为一个索引。

  每个城市索引或者商品索引按频道分为几个type，如下图所示。

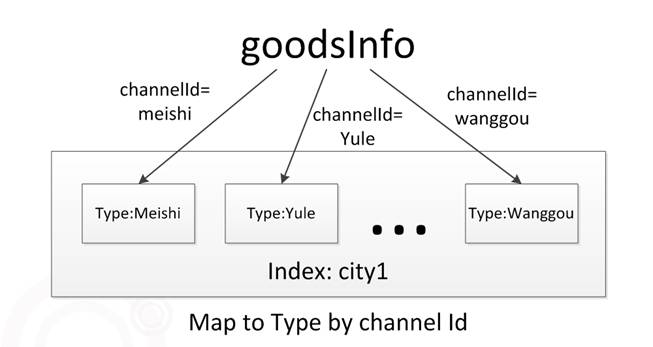


图9 type

   商品频道映射到es的type是很容易理解的，因为每个频道的模型不同：有的频道特有“用餐人数”属性，有的频道特有“出发城市”和“目的地城市”属性 所以每个频道对应一个es的type，每个type绑定一种特定的mapping（这个mapping里面可以指定该频道各自的特殊属性如何储存到ES）。

**6.3.2>门店索引设计**

  门店索引方案，采用了默认的形式，就是一个索引叫做 shop\_index, 5 shard 的形式。

**6.4>集群节点设计方案**

  按照业务拆分，我们将ES拆分为两大集群：商品索引集群（商品分城市索引和全文检索索引）和非商品索引集群（或叫通用集群，目前主要是门店索引和关键词提示索引）。这样分的主要原因是，商品索引数据量较大，而且它是主站主要业务逻辑，所以将其单独设立集群。

  网络拓扑如下图所示：

图10 集群网络拓扑

**6.5>分词设计**

  中文检索最主要的问题是分词，但是分词有一个很大的弊端：当我增加一个新的词库后，需要重新索引现有数据，导致我们重建索引代价较大。所以在牺牲一些查询效率的情况下，窝窝采取了在建立索引时做单字索引，在查询时控制分词索引的方案。

  具体方案如下所示：

图11 分词设计

**6.6>高可用和可伸缩方案**

  看一下窝窝商品索引，窝窝采用的方案是一个城市一个索引，所有索引的“副本（replicas）”都设为 1，这样比如 shop\_index，它有 5 个 shard，每个 shard （只）有一个副本。（注：1个副本一方面可以省空间，另一方面是为了效率，在 ES 0.90版本下，ES 的副本更新是全量备份的方案，多个副本就会有更新效率的问题。ES 1.0 后有改进，王超认为在增加服务器后，可以考虑多增加副本。）

  ES 会保证所有 shard 的主副本不在同一个 node 上面，**但我们是 ES 服务器集群，每台服务器上有多个 node，一个 shard 的主副本不在同一个节点还是不够的**，我们还需要一个 shard 的主副本不在同一台服务器，甚至在多台物理机的情况下保证要保证不在同一个机架上，才可以保证系统的高可用性。

     所以ES提供了一个配置：**cluster.routing.allocation.awareness.attributes: rack\_id**。

     这个属性保证了主副 shard 会分配到名称不同的 rack\_id 上面。

  当我们停止一个节点时，如停止 174\_node\_2，则 ES 会自动重新平衡数据，如下图所示：

图13 重新分布

  即使一台物理机完全 down 掉，我们可以看到其他物理机上的数据是完整的，ES 依然可以保证服务正常。

**七，ES 的几次事故和教训**

**7.1>误删数据**

  ES 的 Web 控制台权限很大，可以删数据。

  有一天，一个开发者需要查询索引 mapping，他用 firefox 的插件访问，结果 Method 默认居然为 DELETE，如下图所示：

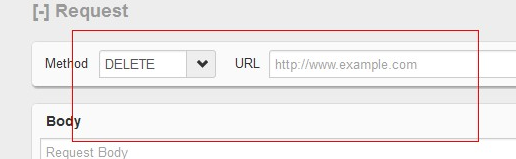


图14 delete

  没有注意，于是悲剧发生了。

教训：

1）后来耀华咨询了长期运维ES的一些人，大部分都建议前置一个 ngnix，通过 ngnix 禁用 delete 和 put 的 HTTP 请求，借此来限制开放的ES接口服务。

2）这次的误操作，实际上**是在没有给定索引的情况下，误执行了DELETE 操作，结果删除了全部索引**。其实配一下 ES 是可以避免的，加入这个配置：

**action.disable\_delete\_all\_indices=true**

   这样会禁止删除所有索引的命令，删除索引的话，必须要给定一个索引 ，稍微安全一些。

**7.2>mvel 脚本引发的ES事故**

**ES集群表象：**

  一天，ES 各个节点负载升高，JVM Full GC 频繁。

  查看其内存使用状况发现，ES 各个节点的 JVM perm 区均处于满或者将要满的状态，如下图所示：

图15 当时perm的容量

 注1：jstat -gc <pid>命令返回结果集中，上图红色方框中字段的含义为：

    PC Current permanent space capacity (KB). 当前perm的容量 ；

    PU Permanent space utilization (KB). perm的使用。

大家可以看到图1中的PU值基本等于PC值了。

**问题原因：**

  头一天上线了商品搜索的一个动态排序功能，它采用 ES 的 [mvel 脚本](http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/modules-scripting.html) 来动态计算商品的排序分值。而 mvel 的原理是基于 JIT，动态字节码生成的，于是很有可能造成 perm 区持续升高，原因是它不断地加载和生成动态 class。

  由于 ES 各个节点的 perm 区接近饱和状态，所以造成了服务器负载升高，GC 频繁，并进一步造成 ES 集群出现了类似于“脑裂”的状态。

**经验教训：**

* 引入新技术，还是要谨慎，毕竟如果真是 mevl 脚本引起的问题，其实线下做压力测试就能提前发现。
* 加强ES的监控。
* 虽然现在回过头来看，如果在第一时间重启所有 nodes，损失应该是最小的——但是王超认为当时采用的保守策略依然是有意义的，因为在弄清楚问题原因之前，直接重启 nodes 有可能反而造成更大的数据破坏。

**7.3>mark shard as failed 的 ES事故**

**问题现象：**

  一天，打算对 JVM 参数和 ES 配置做了小幅度的谨慎调整。

  凌晨 00:10 左右，维护者开始按照计划对 ES 集群的各个 node 依次进行重启操作，以便使新配置的参数生效（这类操作之前进行过很多次，比较熟练）。

1， 使用 http 正常的关闭接口，通知 174\_0 节点进行关闭，成功。

2， 观察其余 node 的状态，几十秒后，ES 剩余的9个节点恢复了 green 状态。

3， 启动 174\_0 节点。

         ——至此仅仅完成了 174\_0 节点的重启工作，但紧接着就发现了问题：174\_0 节点无法加入集群！

  此时的状态是：174\_0 报告自己找不到 master，剩余9个节点的集群依然运行良好。

  于是用 jstat 查看了 174\_0 的内存占用情况，发现其 ParNew 区在正常增长，所以他认为这次重启只是比往常稍慢而已，决定等待。

  但是在 00:16 左右，主节点 174\_4 被踢出集群，174\_3 被选举为主节点。

  之后，日志出现了 shard 损坏的情况：

  [WARN ][cluster.action.shard ] [174\_node\_2] sending failed shard for [goods\_city\_188][0], node[eVxRF1mzRc-ekLi\_4GvoeA], [R], s[STARTED], reason [master [174\_node\_4][6s7o-Yr-QXayxeXRROnFPg][inet[/174:9354]]{rack\_id=rack\_e\_14} marked shard as started, but shard has not been created, mark shard as failed]

  更糟的是，不但损坏的 shard 无法自动回复，而且损坏的 shard 数量越来越多，最终在将近 01:00 的时候，集群由 yellow 状态转为 red 状态，数据不再完整（此时损坏的主 shard 不到 20%，大部分城市还是可以访问的）。

**临时解决办法：**

  首先对主站做业务降级，关闭了来自前端工程的流量。

  维护者开始采用第一套方案：依次关闭所有 node，然后再依次启动所有 node。此时上面新增的 gateway 系列参数开始起作用，集群在达到 6 个 node 以上才开始自动恢复，并且在几分钟后自动恢复了所有的 shard，集群状态恢复 green。

  随后打开了前端流量，主站恢复正常。

  接着补刷了过去2小时的数据到 ES 中。

  至此，故障完全恢复。

**经验教训：**

1， 此次事故发生时，出问题的 nodes 都是老配置；而事故修复之后的所有 nodes 都是采用的新配置，所以**可以确定此次问题并不是新配置导致的**。

2， **ES 自身的集群状态管理（至少在 0.90.2 这个版本上）是有问题的**——先是从正常状态逐渐变为“越来越多的 shard 损坏”，重启之后数据又完全恢复，所有 shard 都没有损坏。

3， 由于是深夜且很短时间就恢复了服务，所幸影响范围较小。

4， **故障原因不明**，所以随后安排 ES 从 0.90 版本升级到 1.3 版本。

**八，ES 存在的问题以及改进**

  Elastic Search 在窝窝运行几年来基本稳定，可靠性和可用性也比较高，但是也暴露了一些问题：

1. ES 的更新效率，作为基于 lucene 的分布式中间件，受限于底层数据结构，所以其更新索引的效率较低，lucene 一直在优化；
2. ES 的可靠性的前提是保证其集群的整体稳定性，但我们遇到的情况，往往是当某个节点性能不佳的情况下，可能会拖累与其同服务器上的所有节点，从而造成整个集群的不稳定。
   * 其实解决这个问题不难，两种方法：
     1. 增加服务器，让节点尽可能地散开；
     2. 当某个节点出现问题的时候，需要我们及早发现处理，不至于拖累整个集群。其实监控一个节点是否正常的方法不难，ES 是基于 JVM 的服务，它出现问题，往往和 GC、和内存有关，所以只要监控其内存达到某个上限就报警即可；
3. 没有一个好的客户端可视化集群管理工具，官方或者主流的可视化管理工具，基本都是基于 ES 插件的，不符合我们的要求，所以需要一款可用的客户端可视化集群管理工具；
4. ES 的升级问题，由于 ES 是一个快速发展的中间件系统，每一次新版本的更新，更改较大，甚至导致我们无法兼容老版本，所以 ES 升级问题是个不小的问题，再加上我们数据量较大，迁移也比较困难。