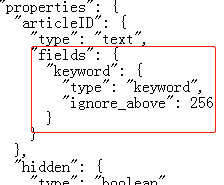
1. type=”text”

默认会设置两个field，一个是field本身，比如articleID，就是分词的；还有一个的话，就是field.keyword，articleID.keyword，默认不分词，会最多保留256个字符.

所以term filter，对text过滤，可以考虑使用内置的field.keyword来进行匹配。但是有个问题，默认就保留256个字符。所以尽可能还是自己去手动建立索引，指定not\_analyzed吧。在最新版本的es中，不需要指定not\_analyzed也可以，将type=keyword即可。

结论：type为text的字段需要建索引时指定为not\_analyzed（或者不用text用keyword类型），才能用term query

1. term和terms

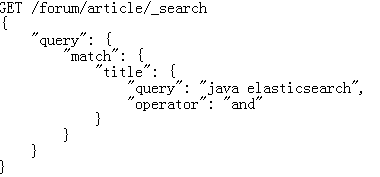
term对搜索文本不分词，直接拿去倒排索引中匹配，根据exact value进行搜索，数字、boolean、date天然支持

terms相当于sql中的in

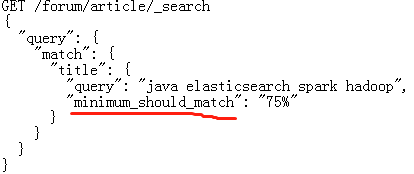
1. filter执行原理深度剖析（bitset机制与caching机制）
2. bool

可以包含must，must\_not，should，组合多个过滤条件。并且，bool可以嵌套。

1. match
   1. 进行full text全文检索.多个条件用空格分隔。如果要检索的field，是not\_analyzed类型的，那么match query也相当于term query.
   2. 如果你是希望所有的搜索关键字都要匹配的，那么就用and，可以实现单纯match query无法实现的效果

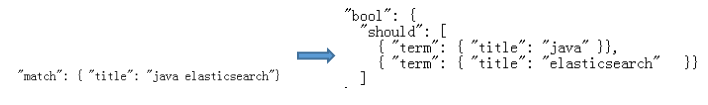
表示搜索的是要这两个字都有的

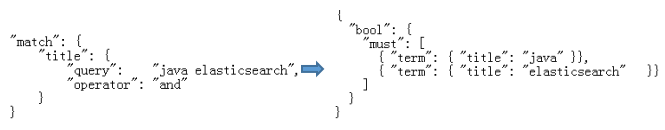
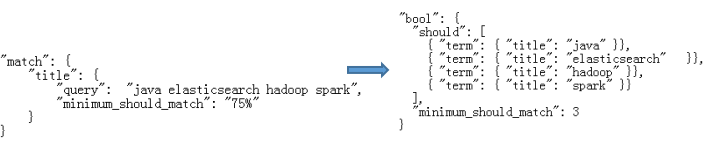
* 1. 指定一些关键字中，必须至少匹配其中的多少个关键字，才能作为结果返回



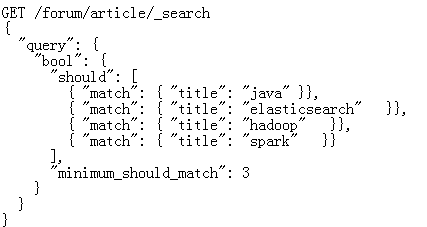
* 1. 底层实现

诸如上面的match query进行多值搜索的时候，es会在底层自动将这个match query转换为bool的语法，bool should，指定多个搜索词，同时使用term query



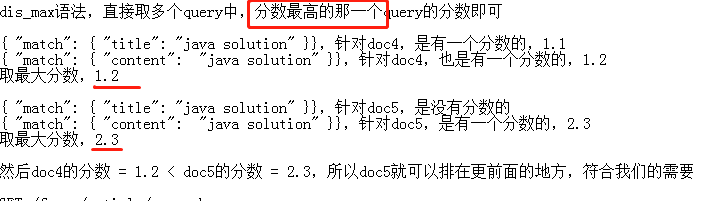
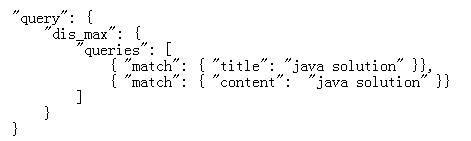
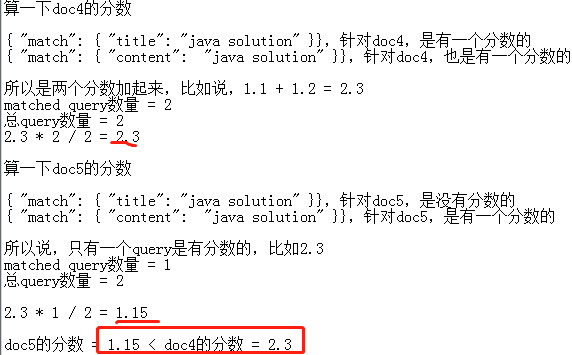
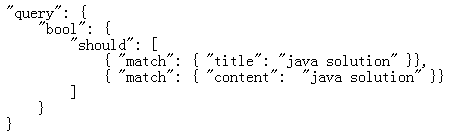
1. should
   1. 默认情况下，should是可以不匹配任何一个的。但是有个例外的情况，如果没有must的话，那么should中必须至少匹配一个才可以（第5.3条的另一种做法）



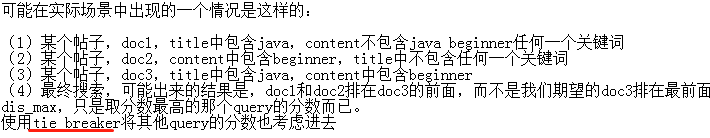
1. best fields策略
   1. best fields策略，就是说，将某一个field匹配尽可能多的关键词的doc优先返回回来.
   2. 实现方式：dis\_max语法，直接取多个query中，分数最高的那一个query的分数即可
   3. 举例说明：基础数据（两个对比）

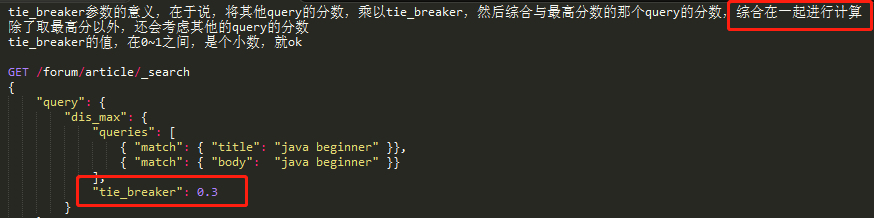
1547521024(1)1547520991(1)

默认情况下计算每个document的relevance score：每个query的分数，乘以matched query数量，除以总query数量。如果用dis\_max,那直接取分数最高的



* 1. tie\_breaker





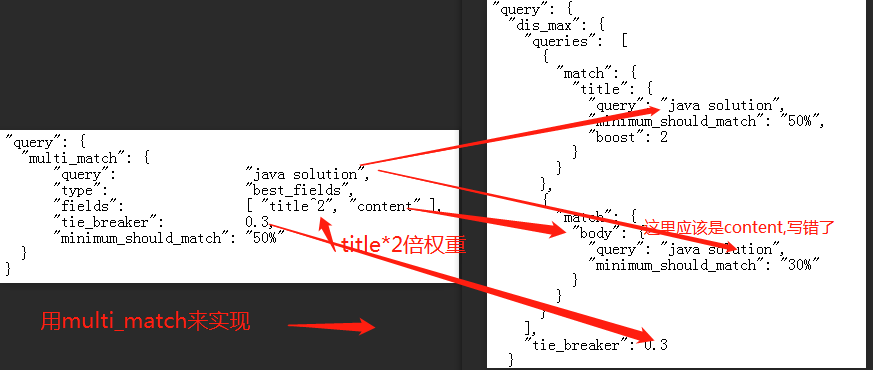
* 1. minimum\_should\_match

去长尾，long tail

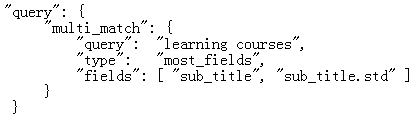
长尾，比如你搜索5个关键词，但是很多结果是只匹配1个关键词的，其实跟你想要的结果相差甚远，这些结果就是长尾

minimum\_should\_match，控制搜索结果的精准度，只有匹配一定数量的关键词的数据，才能返回

* 1. multi\_match语法实现dis\_max+tie\_breaker



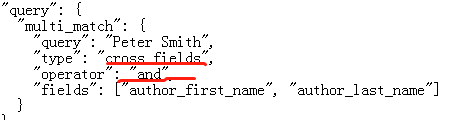
1. most fields策略
   1. 概念：尽可能返回更多field匹配到某个关键词的doc，优先返回回来
   2. 用法



1. cross fields策略
   1. 概念

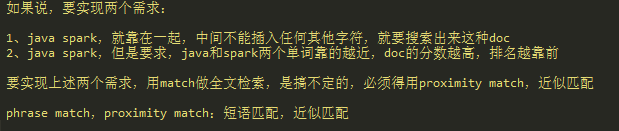
cross-fields搜索，一个唯一标识，跨了多个field。比如一个人，标识，是姓名；一个建筑，它的标识是地址。姓名可以散落在多个field中，比如first\_name和last\_name中，地址可以散落在country，province，city中。

* 1. 方法

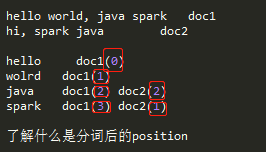


* 1. 优点
     1. 要求Peter必须在author\_first\_name或author\_last\_name中出现并且要求Smith必须在author\_first\_name或author\_last\_name中出现
     2. most\_fields，没办法用minimum\_should\_match去掉长尾数据，就是匹配的特别少的结果 --> 解决，既然每个term都要求出现，长尾肯定被去除掉了

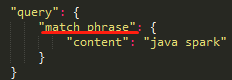
1. 近似匹配
   1. 概念



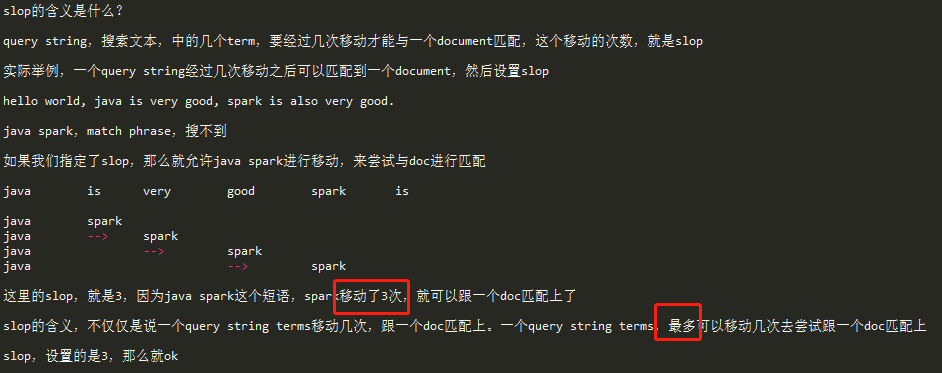
* 1. term position



* 1. match\_phrase
     1. 用法



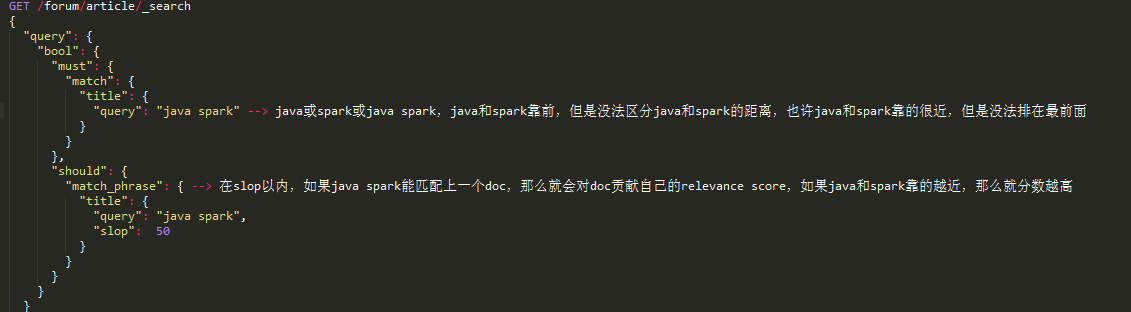
* + 1. 基本原理
       1. 要找到每个term都在的一个共有的那些doc，就是要求一个doc，必须包含每个term，才能拿出来继续计算
       2. 如果有两个单词，那后一个单词的position必须比前一个单词的position大1
  1. slop



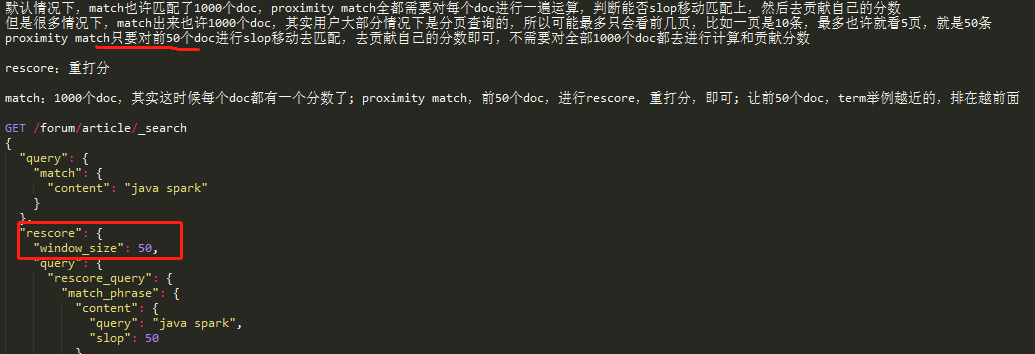


加了slop的phrase match，就是proximity match，近似匹配

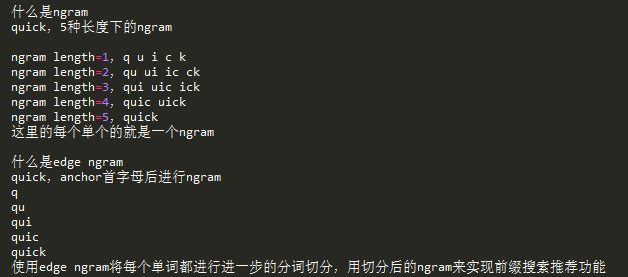
* 1. 使用match和近似匹配实现召回率与精准度的平衡
     1. 召回率:比如你搜索一个java spark，总共有100个doc，能返回多少个doc作为结果，就是召回率，recall
     2. 精准度:比如你搜索一个java spark，能不能尽可能让包含java spark，或者是java和spark离的很近的doc，排在最前面，precision
     3. 近似匹配的时候，召回率比较低，精准度太高了.但是有时可能我们希望的是匹配到几个term中的部分，就可以作为结果出来，这样可以提高召回率。同时我们也希望用上match\_phrase根据距离提升分数的功能，让几个term距离越近分数就越高，优先返回.就是优先满足召回率，意思，java spark，包含java的也返回，包含spark的也返回，包含java和spark的也返回；同时兼顾精准度，就是包含java和spark，同时java和spark离的越近的doc排在最前面.此时可以用bool组合match query和match\_phrase query一起，来实现上述效果



* 1. 使用rescoring机制优化近似匹配搜索的性能

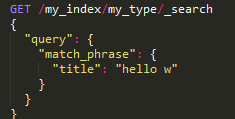


* 1. 通过ngram分词机制实现index-time搜索推荐
     1. ngram

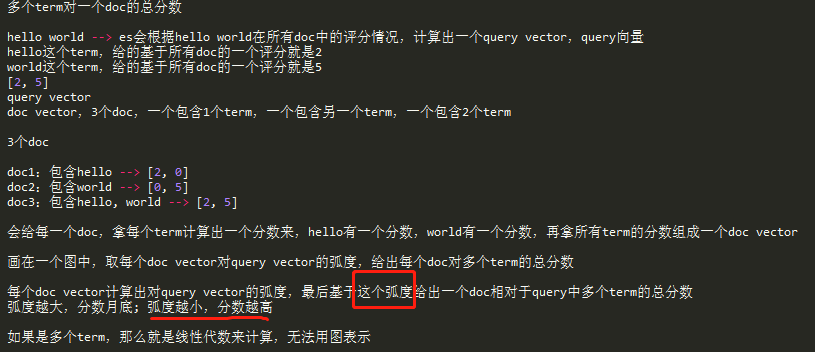
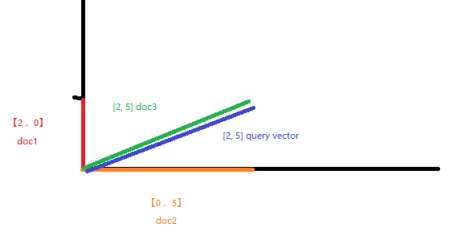


* + 1. 实现index-time搜索推荐

直接在建立索引的时候就建立ngram，这样搜索的时候，不用再根据一个前缀，然后扫描整个倒排索引了; 简单的拿前缀去倒排索引中匹配即可，如果匹配上了，那么就好了; 用match效率更高.

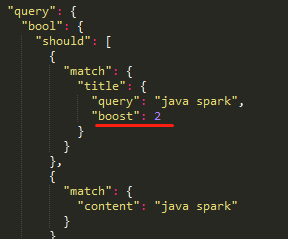


1. 向量空间模型算法(vector space model)



1. 四种常见的相关度分数优化方法
   1. query-time boost

可以增加权重，用的比较多



* 1. 重构查询结构

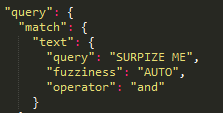
在es新版本中，影响越来越小了。一般情况下，没什么必要的话，大家不用也行

* 1. negative boost
  2. constant\_score

如果你压根儿不需要相关度评分，直接走constant\_score加filter，所有的doc分数都是1，没有评分的概念了

1. fuzzy

fuzzy搜索技术 --> 自动将拼写错误的搜索文本，进行纠正，纠正以后去尝试匹配索引中的数据

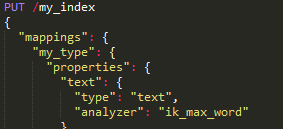


1. IK中文分词
   1. 安装

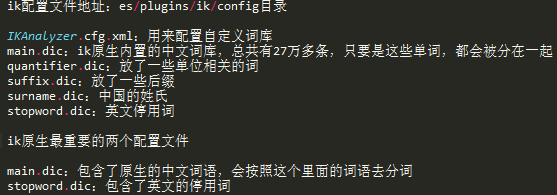
把分词器PLUGIN放到ES/plugin/ik文件夹下，如没有ik文件夹，需要手动建立

* 1. 使用

创建mapping的时候手动指定analyzer,后面就可以用中文分词器来查询了



* 1. 自定义配置



* 1. 修改IK分词器源码来基于mysql热更新词库
     1. 下载源码
     2. 修改ik分词器源码，然后手动支持从mysql中每隔一定时间，自动加载新的词库
     3. mvn package打包代码
     4. 解压缩ik压缩包，将mysql驱动jar，放入ik的目录下
     5. 重启es

1. bucket和metric

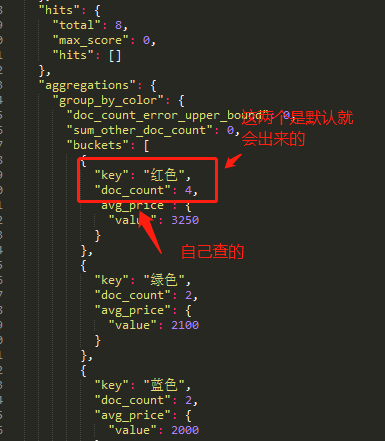
bucket就是group by的字段。metric就是对group by字段进行的操作，计算一个bucket内所有数据的数量，或者计算一个bucket内所有数据的平均值，最大值，最小值

如select count(\*) from access\_log group by user\_id

bucket：group by user\_id --> 那些user\_id相同的数据，就会被划分到一个bucket中

metric：count(\*)，对每个user\_id bucket中所有的数据，计算一个数量

例如：在一个aggs执行的bucket操作（terms），平级的json结构下，再加一个aggs，这个第二个aggs内部，同样取个名字，执行一个metric操作，avg，对之前的每个bucket中的数据的指定的field，price field，求一个平均值

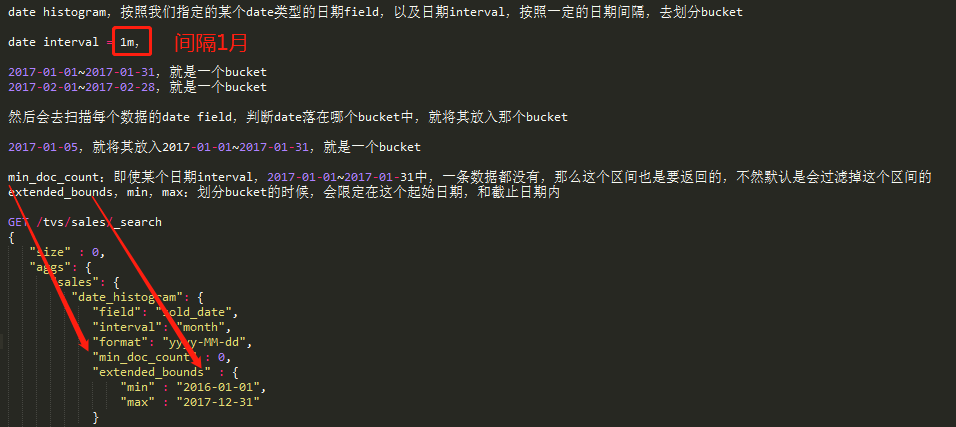


1. histogram

histogram：类似于terms，也是进行bucket分组操作，接收一个field，按照这个field的值的各个范围区间，进行bucket分组操作



1. date histogram



1. 聚合的一些综合操作
   1. 下钻分析

下钻的意思是，已经分了一个组了，比如说颜色的分组，然后还要继续对这个分组内的数据，再分组，比如一个颜色内，还可以分成多个不同的品牌的组，最后对每个最小粒度的分组执行聚合分析操作，这就叫做下钻分析

es，下钻分析，就要对bucket进行多层嵌套，多次分组

按照多个维度（颜色+品牌）多层下钻分析，而且学会了每个下钻维度（颜色，颜色+品牌），都可以对每个维度分别执行一次metric聚合操作



* 1. global

单个品牌与所有品牌销量对比

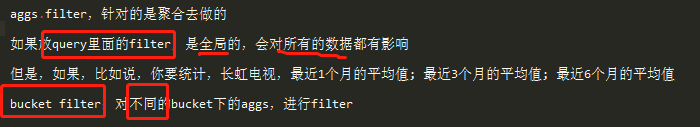


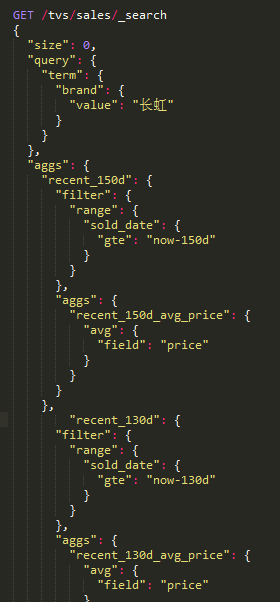
* 1. 过滤+聚合

统计价格大于1200的电视平均价格



* 1. bucket 中进行filter





1. 易并行聚合算法

有些聚合分析的算法，是很容易就可以并行的，比如说max

有些聚合分析的算法，是不好并行的，比如说，count(distinct)，并不是说，在每个node上，直接就出一些distinct value，就可以的，因为数据可能会很多

es会采取近似聚合的方式，就是采用在每个node上进行近估计的方式，得到最终的结论，cuont(distcint)，100万，1050万/95万 --> 5%左右的错误率

近似估计后的结果，不完全准确，但是速度会很快，一般会达到完全精准的算法的性能的数十倍

1. 三角选择原则

精准+实时+大数据 --> 选择2个

* 1. 精准+实时: 没有大数据，数据量很小，那么一般就是单击跑，随便你则么玩儿就可以
  2. 精准+大数据：hadoop，批处理，非实时，可以处理海量数据，保证精准，可能会跑几个小时
  3. 大数据+实时：es，不精准，近似估计，可能会有百分之几的错误率

1. 近似聚合算法

如果采取近似估计的算法：延时在100ms左右，0.5%错误

如果采取100%精准的算法：延时一般在5s~几十s，甚至几十分钟，几小时， 0%错误

1. cartinality
   1. 概念

去重，cartinality metric，对每个bucket中的指定的field进行去重，取去重后的count，类似于count(distcint)

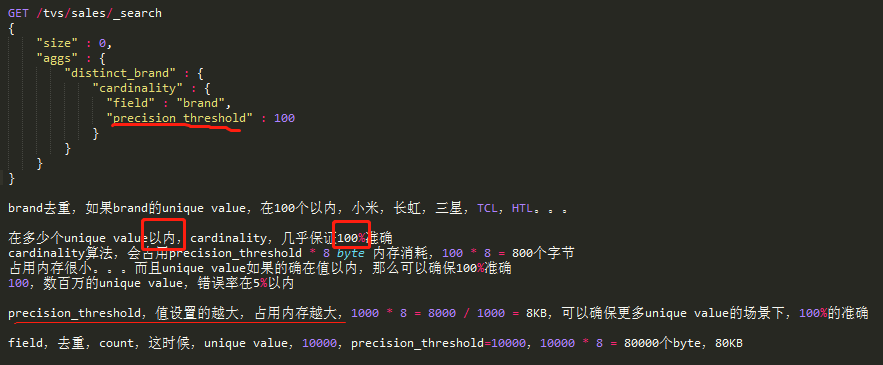
5%的错误率，性能在100ms左右

例：统计1月份有几个brand,2月份有几个brand,3月份有几个brand....



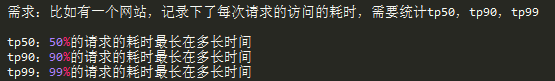
* 1. 优化

precision\_threshold优化准确率和内存开销



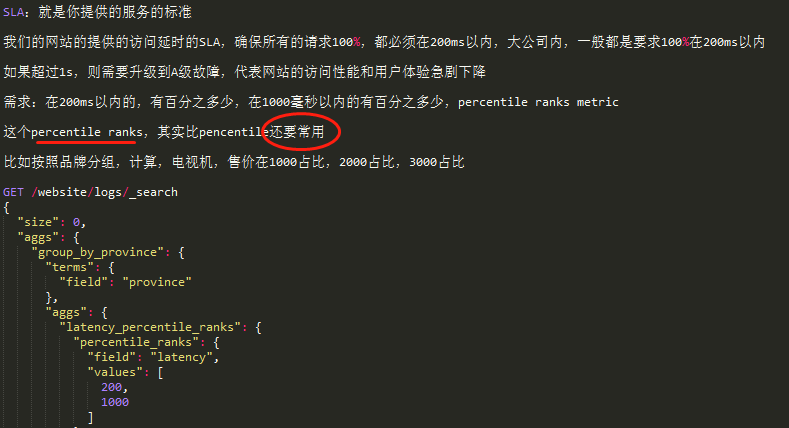
1. percentiles

一般用于计算网站访问延时统计

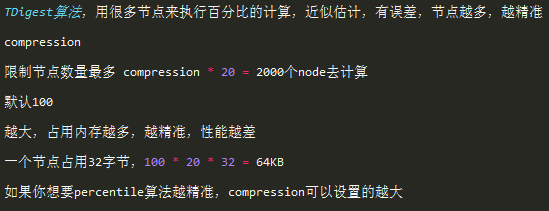




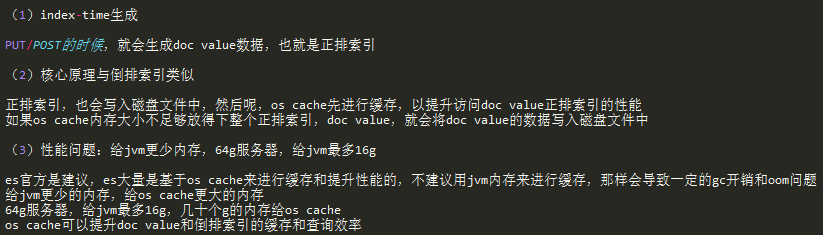
1. percentile ranks
   1. 概念



* 1. 优化

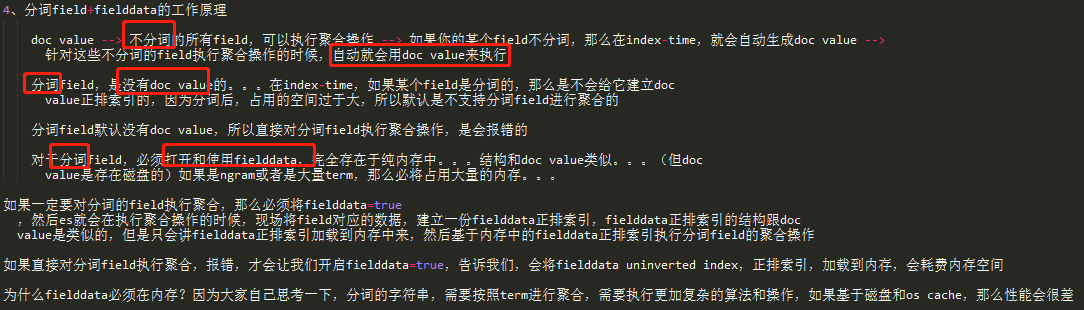


1. 正排索引（doc value）
   1. 概念
   2. 内部原理



1. field data
   1. 概念

对于分词的field执行aggregation，发现报错.对分词的field，直接执行聚合操作，会报错，大概意思是说，你必须要打开fielddata，然后将正排索引数据加载到内存中，才可以对分词的field执行聚合操作，而且会消耗很大的内存.如果要对分词的field执行聚合操作，必须将fielddata设置为true。如果对不分词的field执行聚合操作，直接就可以执行，不需要设置fieldata=true

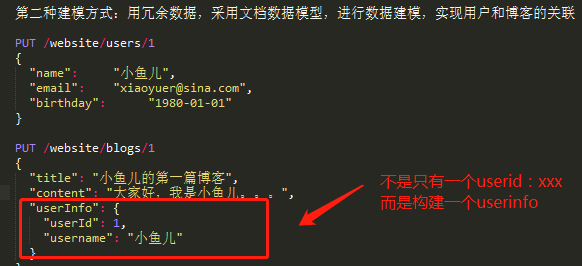


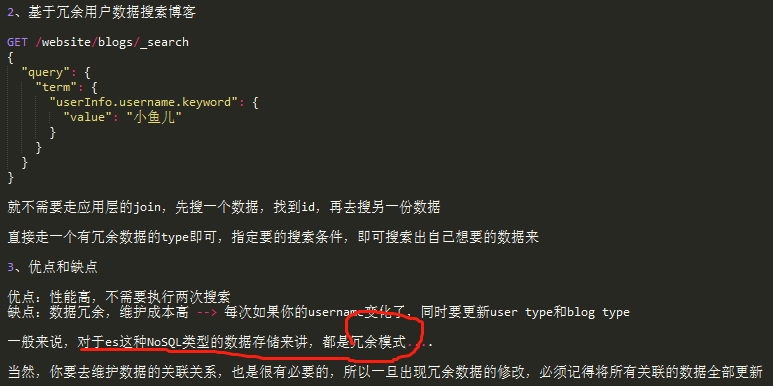
doc value存在于磁盘，在index的时候自动生成

field data存在于内存，并且是filed级别，lazy加载，在对这个field进行聚合时才会生成

**实际应用中，尽量避免对分词的field进行聚合**

1. 数据建模
   1. 冗余思想





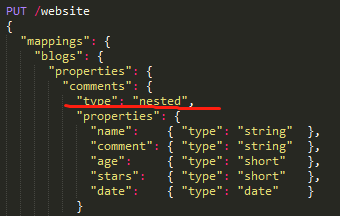
* 1. nested object

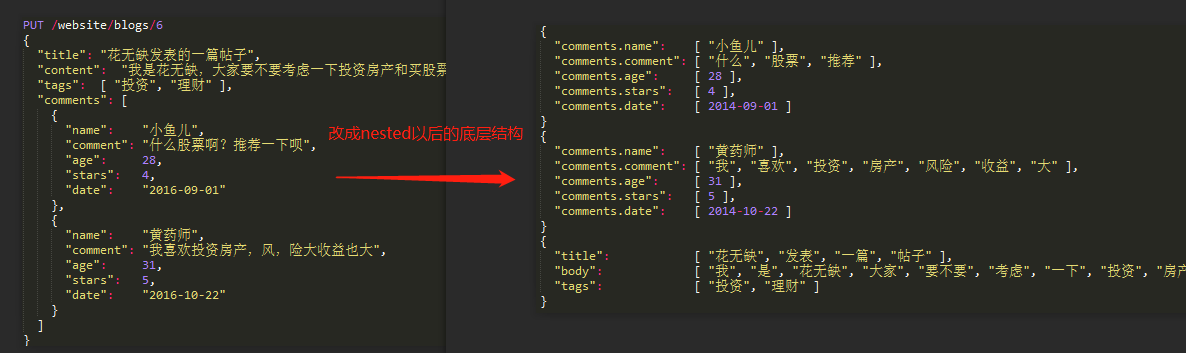
像以上的那种冗余方式建模，object嵌套object



如果搜索被年龄是28岁的黄药师评论过的博客，搜索，会搜索出结果，这个并不是想要的结果

把mapping改一下，把comments这个字段改成nested





搜索的时候外层带上nested,path指明要nested的字段

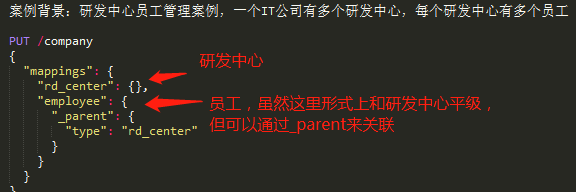


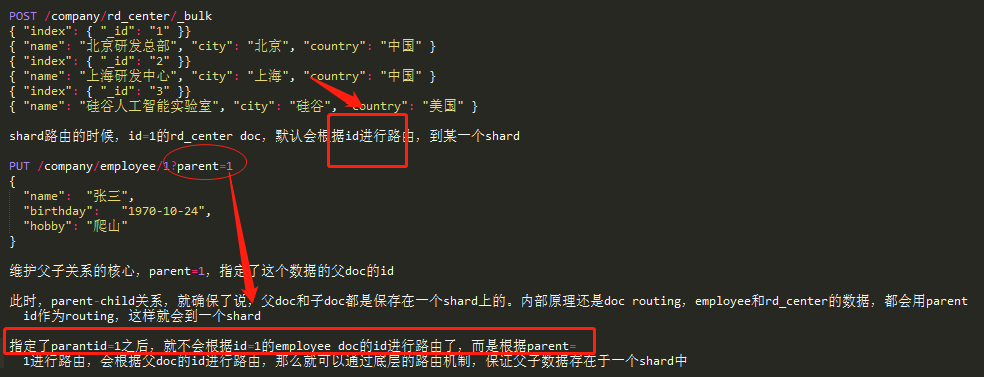
这时如果搜索被年龄是28岁的黄药师评论过的博客，那就搜不出结果了。

* 1. 父子关系数据建模

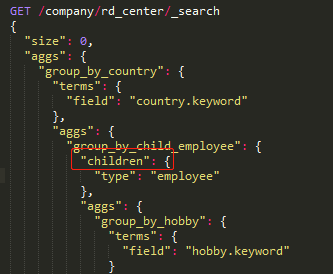
一对多关系的建模，维护起来比较方便，而且我们之前说过，类似关系型数据库的建模方式，应用层join的方式，会导致性能比较差，因为做多次搜索。父子关系的数据模型，不会，性能很好。因为虽然数据实体之间分割开来，但是我们在搜索的时候，由es自动为我们处理底层的关联关系，并且通过一些手段保证搜索性能。

要点：父子关系元数据映射，用于确保查询时候的高性能，但是有一个限制，就是父子数据必须存在于一个shard中



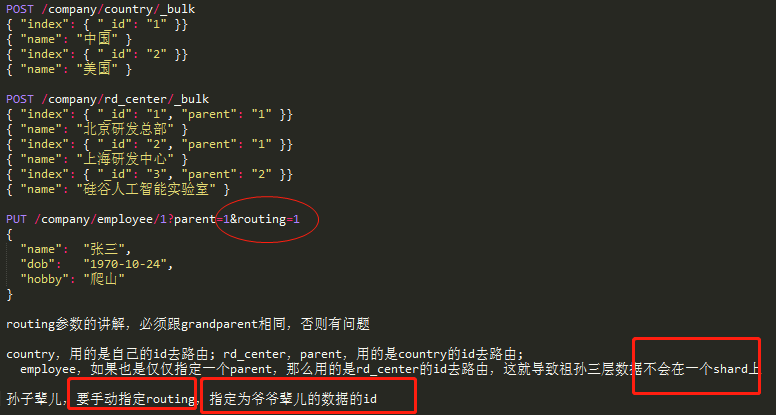


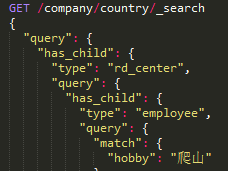
在这种结构下对子数据进行聚合



* 1. 祖孙三层关系的数据建模，搜索



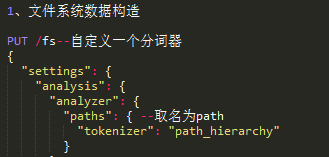




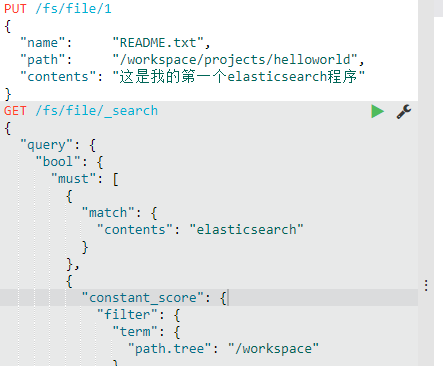
* 1. 文件系统这种的有多层级关系的数据进行建模

用path\_hierarchy

fs :file syste

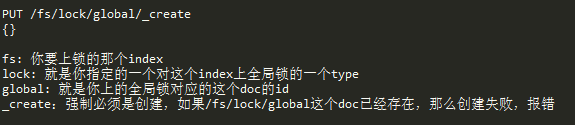






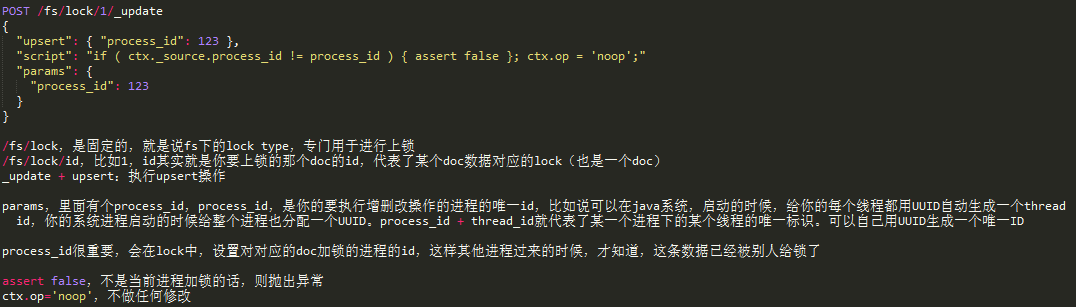
1. 全局锁

直接锁掉整个索引



1. document level锁

是用脚本进行上锁

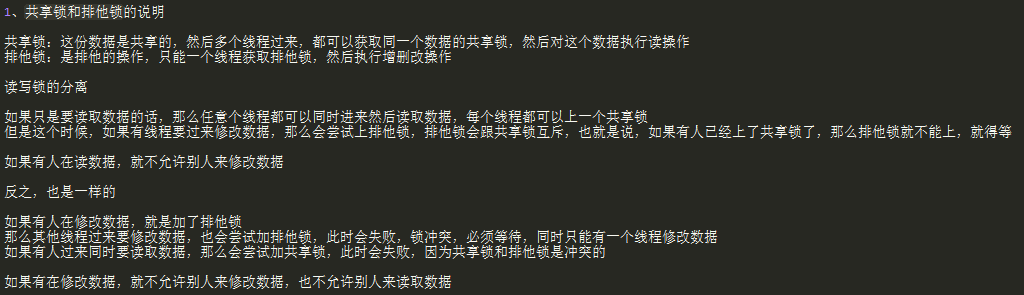


释放锁

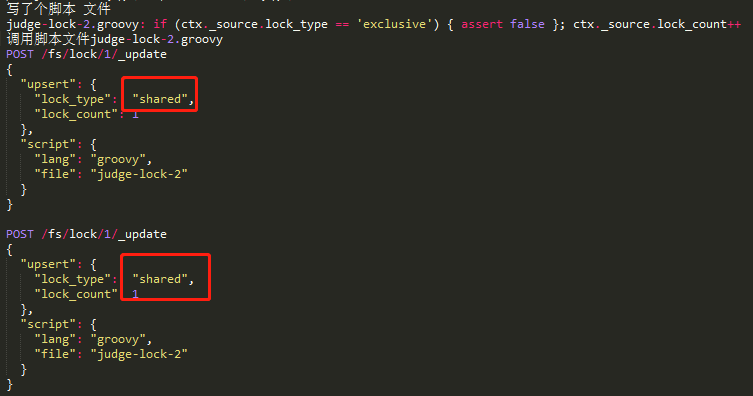
先要POST /fs/\_refresh

再执行PUT /fs/lock/\_bulk{ "delete": { "\_id": 1}}

1. 共享锁和排他锁



* 1. 有人上了共享锁，你还是要上共享锁，直接上就行了，没问题，只是lock\_count加1，共享锁用的是upsert语法(脚本文件写在config/scripts里面，后缀要是groovy)

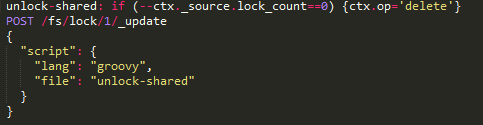


* 1. 已经有人上了共享锁，然后有人要上排他锁

排他锁用的不是upsert语法，用的是create语法，要求lock必须不能存在，直接自己是第一个上锁的人，上的是排他锁

1547797285(1)

* 1. 对共享锁进行解锁



每次解锁一个共享锁，就对lock\_count先减1，如果减了1之后，是0，那么说明所有的共享锁都解锁完了，此时就就将/fs/lock/1删除，就彻底解锁所有的共享锁

* 1. 上排他锁，再上排他锁 ，会报错
  2. 上排他锁，上共享锁，会报错
  3. 解锁排他锁: DELETE /fs/lock/1