在研究了一段时间后，发现ES不适合做为数据存储使用，理由如下：

1. **mapping不可改，不能改index属性**。ES中以定义的mapping不能修改名字和属性，无法修改名字勉强还能接受，但无法修改属性。官方文档中介绍了几种修改mapping的方法。一个是新建一个字段，程序中所有地方修改名字，这对于复杂的项目容易出错，而且无法保留原来的数据；另一个是利用alias创建一个新的索引，但是所有数据需要重新导入，这需要很长时间，操作性不强。
2. ****无法多对多。****ES中提供3种关联关系：Field collapsing（严格来说不是关联），Nested object，Parent-child。前两种都是直接将一个mapping声明在另一个mapping中，第三种关联是在创建子文档是指明他的父文档，但是一个子文档只能有一个父文档，因此也不能实现多对多的关联。其实如果理解了ES的目的是提升检索效率，就不难理解为什么没有多对多关联了，在关系数据库里这就是个效率瓶颈。
3. ****没有用户验证和权限控制。****ES本身的访问权限可以通过nginx进行控制，但是同一个ES中不同索引间目前是没有权限控制的。
4. ****从ES设计的初衷看，为了检索，为了统计。****这个从字段的store属性中可以看出来，查看ES手册（https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-store.html）可以发现，默认情况下字段的原始值是不会被保存的，这跟数据存储是南辕北辙了。
5. ****项目开始时不好确定**shards**数量。****少了以后扩展不方便，多了一开始影响性能。这个可以通过将type命名为doctype-yyyymmdd来解决，每天都生成新的一个或多个shard，但是注意在搜索时需要在doctype-\*中搜索。

****ES非常适合特定的需求，但不适合用于数据存储****。ES索引速度快，扩展方便，性能优异，但在功能上不适合作为数据库使用。数据存储的目的是为了以后能方便的使用，不仅是针对当前的需求，也要为未来可能出现的需求做准备。由于ES有以上几点问题，无法适应需求变化。

ES适合的场景

检索。ES本身作为一个搜索引擎，用来处理检索的任务再合适不过。你可以在线上项目中直接将内容写入ES以提供检索服务，也可以把以往的数据导入ES以处理特定的需求。

统计。ES的统计也是基于检索功能的，聚合功能使得统计结果处理起来非常方便。如果你只需要统计而不用检索，可能有其他工具更适合你，比如Spark SQL。

**详细描述一下Elasticsearch索引文档的过程。**

协调节点默认使用文档ID参与计算（也支持通过routing），以便为路由提供合适的分片。

shard = hash(document\_id) % (num\_of\_primary\_shards)

当分片所在的节点接收到来自协调节点的请求后，会将请求写入到Memory Buffer(客户端不可读)，然后定时（默认是每隔1秒）写入到Filesystem Cache(客户端可读)，这个从Momery Buffer到Filesystem Cache的过程就叫做refresh；

当然在某些情况下，存在Momery Buffer和Filesystem Cache的数据可能会丢失，ES是通过translog的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到translog中，当Filesystem cache中的数据写入到磁盘中时，才会清除掉，这个过程叫做flush；

在flush过程中，内存中的缓冲将被清除，内容被写入一个新段，段的fsync将创建一个新的提交点，并将内容刷新到磁盘，旧的translog将被删除并开始一个新的translog。

flush触发的时机是定时触发（默认30分钟）或者translog变得太大（默认为512M）时；

索引历史数据时，可以关闭refresh，之后再refresh

也就是说，如果在这期间发生异常，Elasticsearch 会从 commit 位置开始，恢复整个 translog 文件中的记录，保证数据一致性。

等到真正把 segment 刷到磁盘，且 commit 文件进行更新的时候， translog 文件才清空。这一步，叫做 flush。同样，Elasticsearch 也提供了 /\_flush 接口。

对于 flush 操作，Elasticsearch 默认设置为：每 30 分钟主动进行一次 flush，或者当 translog 文件大小大于 512MB (老版本是 200MB)时，主动进行一次 flush。这两个行为，可以分别通过 index.translog.flush\_threshold\_period 和 index.translog.flush\_threshold\_size 参数修改。

如果对这两种控制方式都不满意，Elasticsearch 还可以通过 index.translog.flush\_threshold\_ops 参数，控制每收到多少条数据后 flush 一次。

副本配置和分片配置不一样，是可以随时调整的。有些较大的索引，甚至可以在做 forcemerge 前，先把副本全部取消掉，等 optimize 完后，再重新开启副本，节约单个 segment 的重复归并消耗。

# curl -XPUT http://127.0.0.1:9200/logstash-mweibo-2015.05.02/\_settings -d '{

"index": { "number\_of\_replicas" : 0 }

}'

热索引分片不均   
默认情况下，ES 集群的数据均衡策略是以各节点的分片总数(indices\_all\_active)作为基准的。这对于搜索服务来说无疑是均衡搜索压力提高性能的好办法。但是对于 Elastic Stack 场景，一般压力集中在新索引的数据写入方面。正常运行的时候，也没有问题。但是当集群扩容时，新加入集群的节点，分片总数远远低于其他节点。这时候如果有新索引创建，ES 的默认策略会导致新索引的所有主分片几乎全分配在这台新节点上。整个集群的写入压力，压在一个节点上，结果很可能是这个节点直接被压死，集群出现异常。   
所以，对于 Elastic Stack 场景，强烈建议大家预先计算好索引的分片数后，配置好单节点分片的限额。比如，一个 5 节点的集群，索引主分片 10 个，副本 1 份。则平均下来每个节点应该有 4 个分片，那么就配置：

# curl -s -XPUT http://127.0.0.1:9200/logstash-2015.05.08/\_settings -d '{

"index": { "routing.allocation.total\_shards\_per\_node" : "5" }

}'

注意，这里配置的是 5 而不是 4。因为我们需要预防有机器故障，分片发生迁移的情况。如果写的是 4，那么分片迁移会失败。

## **reroute 接口**

上面说的各种配置，都是从策略层面，控制分片分配的选择。在必要的时候，还可以通过 ES 的 reroute 接口，手动完成对分片的分配选择的控制。

reroute 接口支持五种指令：allocate\_replica, allocate\_stale\_primary, allocate\_empty\_primary，move 和 cancel。常用的一般是 allocate 和 move：

allocate\_\* 指令

因为负载过高等原因，有时候个别分片可能长期处于 UNASSIGNED 状态，我们就可以手动分配分片到指定节点上。默认情况下只允许手动分配副本分片(即使用 allocate\_replica)，所以如果要分配主分片，需要单独加一个 accept\_data\_loss选项：

# curl -XPOST 127.0.0.1:9200/\_cluster/reroute -d '{ "commands" : [ { "allocate\_stale\_primary" : { "index" : "logstash-2015.05.27", "shard" : 61, "node" : "10.19.0.77", "accept\_data\_loss" : true } } ] }'

注意，allocate\_stale\_primary 表示准备分配到的节点上可能有老版本的历史数据，运行时请提前确认一下是哪个节点上保留有这个分片的实际目录，且目录大小最大。然后手动分配到这个节点上。以此减少数据丢失。

move 指令

因为负载过高，磁盘利用率过高，服务器下线，更换磁盘等原因，可以会需要从节点上移走部分分片：

curl -XPOST 127.0.0.1:9200/\_cluster/reroute -d '{ "commands" : [ { "move" : { "index" : "logstash-2015.05.22", "shard" : 0, "from\_node" : "10.19.0.81", "to\_node" : "10.19.0.104" } } ] }'

## **节点下线**

集群中个别节点出现故障预警等情况，需要下线，也是 Elasticsearch 运维工作中常见的情况。如果已经稳定运行过一段时间的集群，每个节点上都会保存有数量不少的分片。这种时候通过 reroute 接口手动转移，就显得太过麻烦了。这个时候，有另一种方式：

curl -XPUT 127.0.0.1:9200/\_cluster/settings -d '{

"transient" :{

"cluster.routing.allocation.exclude.\_ip" : "10.0.0.1"

} }'

Elasticsearch 集群就会自动把这个 IP 上的所有分片，都自动转移到其他节点上。等到转移完成，这个空节点就可以毫无影响的下线了。

和 \_ip 类似的参数还有 \_host, \_name 等。此外，这类参数不单是 cluster 级别，也可以是 index 级别。下一小节就是 index 级别的用例。

## **冷热数据的读写分离**

Elasticsearch 集群一个比较突出的问题是: 用户做一次大的查询的时候, 非常大量的读 IO 以及聚合计算导致机器 Load 升高, CPU 使用率上升, 会影响阻塞到新数据的写入, 这个过程甚至会持续几分钟。所以，可能需要仿照 MySQL 集群一样，做读写分离。

### **实施方案**

1、N 台机器做热数据的存储, 上面只放当天的数据。这 N 台热数据节点上面的 elasticsearc.yml 中配置 node.attr.tag: hot

2、之前的数据放在另外的 M 台机器上。这 M 台冷数据节点中配置 node.attr.tag: stale

3、模板中控制对新建索引添加 hot 标签：

{ "order" : 0,

"template" : "\*",

"settings" : {

"index.routing.allocation.include.tag" : "hot"

} }

每天计划任务更新索引的配置, 将 tag 更改为 stale, 索引会自动迁移到 M 台冷数据节点

curl -XPUT http://127.0.0.1:9200/indexname/\_settings -d'

{

"index": {

"routing": {

"allocation": { "include": {

"tag": "stale" } } } } }'

这样，写操作集中在 N 台热数据节点上，大范围的读操作集中在 M 台冷数据节点上。避免了堵塞影响。 该方案运用的，是 Elasticsearch 中的 allocation filter 功能

ES 是一个 P2P 类型(使用 gossip 协议)的分布式系统，在 Elasticsearch 2.0 之前，无阻碍的网络下，所有配置了相同 cluster.name 的节点都自动归属到一个集群中。ES 从 2.0 版本开始，默认的自动发现方式改为了单播(unicast)方式。配置里提供几台节点的地址，ES 将其视作 gossip router 角色，借以完成集群的发现。

network.host: "192.168.0.2"

discovery.zen.minimum\_master\_nodes: 3

discovery.zen.ping\_timeout: 100s #选master时有用

discovery.zen.fd.ping\_timeout: 100s

discovery.zen.fd.ping\_interval: 10s

discovery.zen.fd.ping\_retries: 10

discovery.zen.ping.unicast.hosts: ["10.19.0.97","10.19.0.98","10.19.0.99","10.19.0.100"]

Elasticsearch选主算法

1、对所有可以成为master的节点根据nodeId排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第0位）节点，暂且认为它是master节点。

2、如果对某个节点的投票数达到一定的值（可以成为master节点数n/2+1）并且该节点自己也选举自己，那这个节点就是master。否则重新选举。

3、对于brain split问题，需要把候选master节点最小值设置为可以成为master节点数n/2+1（quorum ）

**elasticsearch-ik分词插件：**https://github.com/medcl/elasticsearch-analysis-ik/releases

**zepplin版本** https://zeppelin.apache.org/supported\_interpreters.html

安装：http://blog.csdn.net/sinat\_28224453/article/details/51134978

**中文权威指南：**

https://www.elastic.co/guide/cn/elasticsearch/guide/current/getting-started.html

https://www.elastic.co/guide/cn/elasticsearch/guide/current/getting-started.html

ES可以使用PUT或者POST对文档进行更新(全部更新)

局部更新，可以添加新字段或者更新已有字段（必须使用POST）

PUT会将新的json值完全替换掉旧的；而POST方式只会更新相同字段的值，其他数据不会改变，新提交的字段若不存在则增加

分词之后查询，并且根据lucene的评分机制(TF/IDF)来进行评分。

get /\_cat/indices?v查看库索引量

GET \_analyze

{

"analyzer": "ik\_max\_word",

"text": "我是JACK"

}

$ curl -XGET 'localhost:9200/startswith/test/\_search?pretty=true&search\_type=dfs\_query\_then\_fetch' -d '{ "query": { "match\_phrase\_prefix": { "title": { "query": "d", "max\_expansions": 5 } } } }'

GET /tx/\_analyze?analyzer=standard&text="a b c"

get /tx/\_analyze?field=baidu\_news.i&text="a b c"

验证查询是否合法：

GET /gb/tweet/\_validate/query

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

只用一个查询字符串，你就可以在一个、多个或者 \_all 索引库（indices）和一个、多个或者所有types中查询：

GET /index\_2014\*/type1,type2/\_search

{}

同时你可以使用 from 和 size 参数来分页：

GET /\_search

{

"from": 30,

"size": 10

}

Sort排序,聚合等要缓存在内存中，修改mapping

**PUT /tx/\_mapping/baidu\_news/ { "properties": { "id": { "type": "text", "fielddata": true } } }**

GET /tx/\_search?pretty

{

"\_source":{

"includes":["id","score"]

} ,

"query":{

"match":{

"content":"信和"

}

},

"sort": [

{

"time": {

"order": "desc"

}}]

}

对于精确值的查询，你可能需要使用 filter 语句来取代 query，因为 filter 将会被缓存

将查询移到 bool 查询的 filter 语句中，这样它就自动的转成一个不评分的 filter 了。

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"range": { "date": { "gte": "2014-01-01" }} IMG_256

}

}

}

输入即搜索（search-as-you-type） 的查询，就是带自动补全的....

在之前的 [前缀查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/prefix-query.html" \o "prefix 前缀查询) 中，我们警告过使用前缀的风险，即 prefix 查询存在严重的资源消耗问题，短语查询的这种方式也同样如此。 前缀 a 可能会匹配成千上万的词，这不仅会消耗很多系统资源，而且结果的用处也不大。

可以通过设置 max\_expansions 参数来限制前缀扩展的影响， 一个合理的值是可能是 50 ：

{

"match\_phrase\_prefix" : {

"brand" : {

"query": "johnnie walker bl",

"max\_expansions": 50

}

}

}

参数 max\_expansions 控制着可以与前缀匹配的词的数量

curl -XPOST 'http://master:9200/tv' -d @tvcount.json

**settings是修改分片和副本数的。**

**mappings是修改字段和类型的。**

索引复制：  
这将把来自twitter索引的文档复制到新的twitter索引中：

POST \_reindex

{

"source": {

"index": "twitter"

},

"dest": {

"index": "new\_twitter"

}

}

**指定type和query**

POST \_reindex

{

"source": {

"index": "twitter",

"type": "\_doc",

"query": {

"term": {

"user": "kimchy"

}

}

},

"dest": {

"index": "new\_twitter"

}

}

创建索引：

\_source 是否存储原始字段

\_all 是否把字段存入\_all大字段

DELETE test PUT test PUT test/test/\_mapping

{ "test":

{ "\_source":

{ "enabled": false },

"properties":

{ "title": { "type": "string",

"index": "not\_analyzed",

"store": "true" },

"content": { "type": "string" } } }

}

**创建索引**

curl -XPUT hadoop-5:9200/testIndex -d '{

"settings":{

"index": { "refresh\_interval" : "-1", //减少shard刷新间隔，用于大规模bulk插入，且对实时查询不要求时使用，完成bulk插入后再修改为1s

"number\_of\_shards": "9", //**初始化9个主分片,分片数量确定后不可修改,非常重要**

"translog.flush\_threshold\_size": "1024mb", //默认是200mb,增大这个参数可以允许translog在flush前存放更大的段(segment);更大的段的创建会减少flush的频率，并且更大的段合并越少，会减少磁盘IO，索引性能更高。

"number\_of\_replicas": "1", //设置1个备份，bulk导入大量的数据，可以考虑不要副本，设置为0

"max\_result\_window": 999999

}

},

"mappings" :{

"testType" : { "include\_in\_all": false, //每个索引都有\_all,把各个field存储进去，禁用\_all,可以在具体字段打开

"dynamic": "true",

"\_source": {

"enabled": "true" //是否存储原始字段

},

"\_all": { "analyzer": "ik\_max\_word" },

"properties": {

"id" : { "type": "keyword",//5x后废弃了string，改为keyword和text

"index": "not\_analyzed",

"include\_in\_all": true,//6.x已废弃，使用copy\_to

"store": true

}

}

}

}

}'

POST edemo/test/\_delete\_by\_query?conflicts=proceed

{

"query": {

"match\_all": {}

}

}

查看：

curl -XGE http://192.168.80.10:9200/zhouls/\_settings?pretty

操作不存在索引：

curl -XPUT '192.168.80.10:9200/liuch/' -d '{"settings":{"number\_of\_shards":3,"number\_of\_replicas":0}}'

操作已存在索引：

curl -XPUT '192.168.80.10:9200/zhouls/\_settings' -d '{"index":{"number\_of\_replicas":1}}'

curl -XPUT 'hadoop-5:9200/testindex/\_settings' -d'{"index":{"refresh\_interval":"1"}}'

字符串将默认被同时映射成text和keyword类型. 例如对下面的文档进行索引后:

{

"foo": "bar"

}

ElasticSearch将会为你创建下面的动态映射(dynamic mappings):

{

"foo": {

"type": "text",//将被分词

"fields": {

"keyword": {

"type": "keyword",//不被分词

"ignore\_above": 256

}

}

}

}

当然, 基于这个映射你即可以在foo字段上进行全文搜索, 也可以通过foo.keyword字段实现关键词搜索及数据聚合.

**查询：**

## **match**

最简单的一个match例子：

查询和"我的宝马多少马力"这个查询语句匹配的文档。

{

"query": {

"match": {

"content" : {

"query" : "我的宝马多少马力"

}

}

}

}

上面的查询匹配就会进行分词，比如"宝马多少马力"会被分词为"宝马 多少 马力", 所有有关"宝马 多少 马力", 那么所有包含这三个词中的一个或多个的文档就会被搜索出来。  
并且根据lucene的评分机制(TF/IDF)来进行评分。

## **match\_phrase**

比如上面一个例子，一个文档"我的保时捷马力不错"也会被搜索出来，那么想要精确匹配所有同时包含"宝马 多少 马力"的文档怎么做？就要使用 match\_phrase 了

{

"query": {

"match\_phrase": {

"content" : {

"query" : "我的宝马多少马力"

}

}

}

}

完全匹配可能比较严，我们会希望有个可调节因子，少匹配一个也满足，那就需要使用到slop。

{

"query": {

"match\_phrase": {

"content" : {

"query" : "我的宝马多少马力",

"slop" : 1

}

}

}

}

## **multi\_match**

如果我们希望两个字段进行匹配，其中一个字段有这个文档就满足的话，使用multi\_match

{

"query": {

"multi\_match": {

"query" : "我的宝马多少马力",

"fields" : ["title", "content"]

}

}

}

## **我们希望完全匹配的文档占的评分比较高，则需要使用best\_fields**

{

"query": {

"multi\_match": {

"query": "我的宝马发动机多少",

"type": "best\_fields",

"fields": [

"tag",

"content"

],

"tie\_breaker": 0.3

}

}

}

意思就是完全匹配"宝马 发动机"的文档评分会比较靠前，如果只匹配宝马的文档评分乘以0.3的系数

## **我们希望越多字段匹配的文档评分越高，就要使用most\_fields**

{

"query": {

"multi\_match": {

"query": "我的宝马发动机多少",

"type": "most\_fields",

"fields": [

"tag",

"content"

]

}

}

}

## **我们会希望这个词条的分词词汇是分配到不同字段中的，那么就使用cross\_fields**

{

"query": {

"multi\_match": {

"query": "我的宝马发动机多少",

"type": "cross\_fields",

"fields": [

"tag",

"content"

]

}

}

}

# **term**

**term是代表完全匹配，即不进行分词器分析，文档中必须包含整个搜索的词汇**

{

"query": {

"term": {

"content": "汽车保养"

}

}

}

查出的所有文档都包含"汽车保养"这个词组的词汇。

使用term要确定的是这个字段是否“被分析”(analyzed)，默认的字符串是被分析的。

拿官网上的例子举例：

mapping是这样的：

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"full\_text": {

"type": "string"

},

"exact\_value": {

"type": "string",

"index": "not\_analyzed"

}

}

}

}

}

PUT my\_index/my\_type/1

{

"full\_text": "Quick Foxes!",

"exact\_value": "Quick Foxes!"

}

其中的full\_text是被分析过的，所以full\_text的索引中存的就是[quick, foxes]，而extra\_value中存的是[Quick Foxes!]。

那下面的几个请求：

GET my\_index/my\_type/\_search

{

"query": {

"term": {

"exact\_value": "Quick Foxes!"

}

}

}

请求的出数据，因为完全匹配

GET my\_index/my\_type/\_search

{

"query": {

"term": {

"full\_text": "Quick Foxes!"

}

}

}

请求不出数据的，因为full\_text分词后的结果中没有[Quick Foxes!]这个分词。

curl -XGET hadoop-5:9200/testindex/jdbc/\_search?pretty&website:网易

term主要用于精确匹配哪些值，比如数字，日期，布尔值或 not\_analyzed 的字符串(未经分析的文本数据类型)：

terms 跟 term 有点类似，但 terms 允许指定多个匹配条件。 如果某个字段指定了多个值，那么文档需要一起去做匹配：

完整的例子，所有http的状态是 302 、304 的， 由于ES中状态是数字类型的字段，所有这里我们可以直接这么写。：

{

"query": {

"terms": {

"status": [

304,

302

]

}

}

}

range过滤允许我们按照指定范围查找一批数据：

{

"range": {

"age": {

"gte": 20,

"lt": 30

}

}

}

范围操作符包含：

gt :: 大于

gte:: 大于等于

lt :: 小于

lte:: 小于等于

exists 和 missing 过滤可以用于查找文档中是否包含指定字段或没有某个字段，类似于SQL语句中的IS\_NULL条件.

{

"exists": {

"field": "title"

}

}

这两个过滤只是针对已经查出一批数据来，但是想区分出某个字段是否存在的时候使用。

**bool过滤**可以用来合并多个过滤条件查询结果的布尔逻辑，它包含一下操作符：

must :: 多个查询条件的完全匹配,相当于 and。

must\_not :: 多个查询条件的相反匹配，相当于 not。

should :: 至少有一个查询条件匹配, 相当于 or。

这些参数可以分别继承一个过滤条件或者一个过滤条件的数组：

{

"bool": {

"must": { "term": { "folder": "inbox" }},

"must\_not": { "term": { "tag": "spam" }},

"should": [

{ "term": { "starred": true }},

{ "term": { "unread": true }}

]

}

}

**match\_all 查询**

可以查询到所有文档，是没有查询条件下的默认语句。

{

"match\_all": {}

}

**count**

GET /twitter/\_doc/\_count

{

"query" : {

"term" : { "user" : "kimchy" }

}

}

**match 查询**

match查询是一个标准查询，不管你需要全文本查询还是精确查询基本上都要用到它。

如果你使用 match 查询一个全文本字段，它会在真正查询之前用分析器先分析match一下查询字符：

{

"match": {

"tweet": "About Search"

}

}

如果用match下指定了一个确切值，在遇到数字，日期，布尔值或者not\_analyzed 的字符串时，它将为你搜索你给定的值：

{ "match": { "age": 26 }}

{ "match": { "date": "2014-09-01" }}

{ "match": { "public": true }}

{ "match": { "tag": "full\_text" }}

提示： **做精确匹配搜索时，你最好用过滤语句，因为过滤语句可以缓存数据。**

match查询只能就指定某个确切字段某个确切的值进行搜索，而你要做的就是为它指定正确的字段名以避免语法错误。

**multi\_match**查询允许你做match查询的基础上同时搜索多个字段，在多个字段中同时查一个：

{

"multi\_match": {

"query": "full text search",

"fields": [ "title", "body" ]

}

}

**bool 查询**

bool 查询与 bool 过滤相似，用于合并多个查询子句。不同的是，bool 过滤可以直接给出是否匹配成功， 而bool 查询要计算每一个查询子句的 \_score （相关性分值）。

must:: 查询指定文档一定要被包含。

must\_not:: 查询指定文档一定不要被包含。

should:: 查询指定文档，有则可以为文档相关性加分。

以下查询将会找到 title 字段中包含 "how to make millions"，并且 "tag" 字段没有被标为 spam。 如果有标识为 "starred" 或者发布日期为2014年之前，那么这些匹配的文档将比同类网站等级高：

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }},

{ "range": { "date": { "gte": "2014-01-01" }}}

]

}

}

{

"query": {

"regexp": {

"hostname": "wxopen.\*"

}

}

}

以什么字符开头的，可以更简单地用 prefix，如下面的例子：

{

"query": {

"prefix": {

"hostname": "wxopen"

}

}

}

单条插入-指定id

curl -XPOST 'localhost:9200/customer/external/1?pretty' -d' {"name": "John Doe" }'

批量插入数据

curl -XPOST 'localhost:9200/bank/account/\_bulk?pretty' --data-binary "@accounts.json"

删除数据：下面的语句将执行删除Customer中ID为2的数据

curl -XDELETE 'localhost:9200/customer/external/2?pretty'

curl -XPOST 'localhost:9200/customer/external/\_delete\_by\_query?pretty' -d '{

"query": {

"match": {

"name": "John"

}

}

}'

 删除全部

{

"query": {

"match\_all": {}

}

}

更新文档: 修改id=1的name属性，并直接增加属性和属性值

curl -XPOST 'localhost:9200/customer/external/1/\_update?pretty' -d ' {

"doc": {

"name": "xyd",

"age": 20

}

}'

更新索引–脚本方式

curl -XPOST 'localhost:9200/customer/external/1/\_update?pretty' -d' {

"script": "ctx.\_source.age += 5"

}'

shard的移动

curl -XPOST 'localhost:9200/\_cluster/reroute' -d '{

"commands" : [ {

"move" :

{

"index" : "test\_aa\_20160529", "shard" : 4,

"from\_node" : "node1", "to\_node" : "node2"

}

},

{

"allocate" :

{

"index" : "test", "shard" : 1, "node" : "node3"

}

}

]

}'

# **ElasticSearch 中文同义词实现**

本文前提默认已经装好ES和IK插件。

1：elasticserach.yml 最后一行添加如下内容（该文件位于elasticsearch-x.x.x/config目录下）：   
index.analysis.analyzer.default.type: ik

2：在elasticsearch-x.x.x/config目录下新建同义词文件synonyms.txt。   
其中，synonyms.txt 编码格式为’utf-8’，内容建议为空。

3：创建索引

curl -XPUT localhost:9200/test -d'{

"settings": {

"index": {

"**analysis**": {

"analyzer": {

"jt\_cn": {

"type": "custom",

"use\_smart": "true",

"tokenizer": "ik\_smart",

"filter": ["jt\_tfr","jt\_sfr"],

"char\_filter": ["jt\_cfr"]

},

"ik\_smart": {

"type": "ik",

"use\_smart": "true"

},

"ik\_max\_word": {

"type": "ik",

"use\_smart": "false"

}

},

"**filter**": {

"jt\_tfr": {

"type": "stop",

"stopwords": [" "]

},

"jt\_sfr": {

"type": "synonym",

"synonyms\_path": "synonyms.txt"

}

},

"char\_filter": {

"jt\_cfr": {

"type": "mapping",

"mappings": [

"| => \\|"

]

}

}

}

}

}

}'

4：创建映射

curl -X PUT localhost:9200/test/haizhi/\_mapping -d '{

"haizhi": {

"properties": {

"title": {

"include\_in\_all": true,

"analyzer": "jt\_cn",

"term\_vector": "with\_positions\_offsets",

"boost": 8,

"store": true,

"type": "string"

}

}

}

}'

5：插入数据

curl -XPUT localhost:9200/test/haizhi/1 -d '{

"title": "番茄"

}'

curl -XPUT localhost:9200/test/haizhi/2 -d '{

"title": "西红柿"

}'

curl -XPUT localhost:9200/test/haizhi/3 -d '{

"title": "我是西红柿"

}'

curl -XPUT localhost:9200/test/haizhi/4 -d '{

"title": "我是番茄"

}'

curl -XPUT localhost:9200/test/haizhi/5 -d '{

"title": "土豆"

}'

curl -XPUT localhost:9200/test/haizhi/6 -d '{

"title": "aa"

}'

6：查询1

curl -XPOST 'localhost:9200/test/haizhi/\_search?pretty' -d '

{

"query": {

"match\_phrase": {

"title": {

"query": "西红柿",

"analyzer": "jt\_cn"

}

}

},

"highlight": {

"pre\_tags": [

"<tag1>",

"<tag2>"

],

"post\_tags": [

"</tag1>",

"</tag2>"

],

"fields": {

"title": {}

}

}

}'

7：查询2   
我们知道“西红柿”和“番茄”是同义词，我们在同义词词典（synonyms.txt）中添加如下内容，并重启ES，再用第6步的查询。

#Example:

西红柿, 番茄

8：查询3   
修改同义词词典（synonyms.txt）为如下内容，重启ES。

#Example:

西红柿, 番茄

超级土豆, 土豆

查询如下：

curl -XPOST 'localhost:9200/test/haizhi/\_search?pretty' -d '

{

"query": {

"match\_phrase": {

"title": {

"query": "超级土豆",

"analyzer": "jt\_cn"

}

}

}

}'

查不到结果

9：查询4   
在{plugins}/elasticsearch-analysis-ik-\*/config/custom/mydict.dic词典中新加“超级土豆”一词，并且重启ES。   
- 结果如下，成功用“超级土豆”搜索到“土豆”一词

10：小结

同义词字典或是IK用户自定义词典更新，必须每次重启elasticsearch才有效。

同义词词对是必须能被完成切分的词语。   
比如在synonyms.txt 文件中增加同义词对： ‘超级土豆’ – ‘土豆’ 。但在实际的搜索中用“超级土豆”是搜不到“土豆”的。因为“超级土豆”会被切分为多个词语。必须在{plugins}/elasticsearch-analysis-ik-\*/config/custom/mydict.dic词典中新加“超级土豆”一词，才能用“超级土豆”一词搜出“土豆”。

**完全匹配**

实现思路，****字段按一字一词的形式分词，再利用短语查询来搜索****。

ES中，可以实现一字一词的的分词器是NGram。

Ngram分词的官方文档地址： https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-ngram-tokenizer.html

它其实是一个上下文相连续字符的分词工具，可以看官方文档中的例子。当我们将它 min\_gram 和 max\_gram都设为1时，它会按一字一词的形式分词。比如“shinyke@189.cn”，分词的结果是["s" , "h" , "i" , "n" , "y" , "k" , "e" , "@" , "1" , "8" , "9" , "." , "c" , "n" ]。

{

  "settings": {

    "analysis": {

      "analyzer": {

        "charSplit": {

          "type": "custom",

              "tokenizer": "ngram\_tokenizer"

        }

      },

     "tokenizer": {

           "ngram\_tokenizer": {

             "type": "nGram",

             "min\_gram": "1",

             "max\_gram": "1",

             "token\_chars": [

               "letter",

               "digit",

               "punctuation"

             ]

           }

        }

      }

   }

}

以上语句中，构建了一个名为“charSplit”的分析器。它使用一个名为“ngram\_tokenizer”的Ngram分词器。

可以用如下语句测试charSplit分析器，可以看到一字一词的效果：

curl -POST http://IP:port/{index\_name}/\_analyze?pretty&analyzer=charSplit

"测试语句"

把这个分析器在mapping里用起来：

"sender": {

   "type": "string",

   "store": "yes",

   "analyzer": "charSplit",

   "fields": {

     "raw": {

       "type": "string",

       "index": "not\_analyzed"

     }

   },

  ...

接下来就可以用match\_phrase来实现完全匹配查询。

/{index\_name}/{type\_name}/\_search

{

  "query": {

    "multi\_match": {

      "query": "@189.cn",

      "type": "phrase", //type指定为phrase，有顺序

      "slop": 0,        //slop指定每个相邻词之间允许相隔多远。此处设置为0，以实现完全匹配。

      "fields": [

        "sender"

      ],

      "analyzer": "charSplit", //分析器指定为charSplit

      "max\_expansions": 1

    }

  },

  "highlight": {   //测试高亮是否正常

    "pre\_tags": [

      "<b>"

    ],

    "post\_tags": [

      "</b>"

    ],

    "fragment\_size": 100,

    "number\_of\_fragments": 2,

    "require\_field\_match": true,

    "fields": {

      "sender": {}

    }

  }

}

phrase查询原始的作用是用来做短语查询，****它有一个重要的特点：有顺序****。我们利用了它匹配的有序性，限制slop为0，则可实现完全匹配查询。

该参数允许你转化一个简单的映射如：

"tweet": {

"type": "string",

"analyzer": "english"

}

为一个多字段映射如：

"tweet": { IMG_256

"type": "string",

"analyzer": "english",

"fields": {

"raw": { IMG_257

"type": "string",

"index": "not\_analyzed"

}

}

}

|  |  |
| --- | --- |
| [IMG_258](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html#CO28-1) | tweet 主字段与之前的一样: 是一个 analyzed 全文字段。 |
| [IMG_259](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html#CO28-2) | 新的 tweet.raw 子字段是 not\_analyzed. |

现在，至少只要我们重新索引了我们的数据，使用 tweet 字段用于搜索，tweet.raw 字段用于排序：

GET /\_search

{

"query": {

"match": {

"tweet": "elasticsearch"

}

},

"sort": "tweet.raw"

}

多字段，设置优先级

GET /\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "War and Peace" }},//贡献评分的1/3,可通过{“title”:{“query”:”war and peace”,”boost”:2}}来提高评分占比

{ "match": { "author": "Leo Tolstoy" }},//贡献评分的1/3

{ "bool": {//贡献评分的1/3,若将traunslator与title并立，则贡献评分1/2

"should": [

{ "match": { "translator": "Constance Garnett" }},

{ "match": { "translator": "Louise Maude" }}

]

}}

]

}

}

}

kibana.yml

server.port: 5601

server.host: 0.0.0.0

server.name: kibana

elasticsearch.url: "http://hadoop-5:9200"

cluster.name: es-tx

node.name: node1

bootstrap.memory\_lock: false

bootstrap.system\_call\_filter: false

network.host: hadoop-5

http.port: 9200

node.master: true

node.data: true

path.data: /data/node1/es

path.logs: /data/node1/eslogs

discovery.zen.fd.ping\_timeout: 100s

discovery.zen.minimum\_master\_nodes: 2

discovery.zen.ping\_timeout: 100s

discovery.zen.ping.unicast.hosts: ["10.167.2222.105","10.167.222.106","10.167.222.107"]

http.cors.enabled: true

http.cors.allow-origin: "\*"

集群配置：

默认情况下，每个节点都有成为主节点的资格，也会存储数据，还会处理客户端的请求。在一个生产集群中我们可以对这些节点的职责进行划分。

　　建议集群中设置 **3台** 以上的节点作为 master 节点【node.master: true node.data: falsenode.ingest:false】，这些节点只负责成为主节点，维护整个集群的状态。  
　　再根据数据量设置一批 data节点【node.master: false node.data: truenode.ingest:false】，这些节点只负责存储数据，后期提供建立索引和查询索引的服务，这样的话如果用户请求比较频繁，这些节点的压力也会比较大  
　　所以在集群中建议再设置一批 ingest 节点也称之为 client 节点【node.master: false node.data: false node.ingest:true】，这些节点只负责处理用户请求，实现请求转发，负载均衡等功能。

　　master节点：普通服务器即可(CPU 内存 消耗一般)  
　　data   节点：主要消耗磁盘，内存  
　　client | ingest  节点：普通服务器即可(如果要进行分组聚合操作的话，建议这个节点内存也分配多一点)

**用户管理：**

新增用户:

POST /\_xpack/security/user/my\_user

{

"password" : "my\_user@123",

"roles" : [ "superuser", "other\_role1" ]

}

查询用户

GET /\_xpack/security/user/my\_user

重置密码

PUT /\_xpack/security/user/my\_user/\_password

{

"password" : "elastic123"

}

禁用my\_user：PUT /\_xpack/security/user/my\_user/\_disable

启用my\_user：PUT /\_xpack/security/user/my\_user/\_enable

删除用户my\_user：DELETE /\_xpack/security/user/my\_user

GET \_xpack/security/user/\_has\_privileges

{

"cluster": [ "monitor", "manage" ],

"index" : [

{

"names": [ "suppliers", "products" ],

"privileges": [ "read" ]

},

{

"names": [ "inventory" ],

"privileges" : [ "read", "write" ]

}

]

}

创建角色

POST /\_xpack/security/role/my\_admin\_role

{

"cluster": ["all"],

"indices": [

{

"names": [ "index1", "index2" ],

"privileges": ["all"],

"field\_security" : { // optional

"grant" : [ "title", "body" ]

},

"query": "{\"match\": {\"title\": \"foo\"}}" // optional

}

],

"run\_as": [ "other\_user" ], // optional

"metadata" : { // optional

"version" : 1

}

}

Setting建好之后只能改副本数，不能改分片数

Mapping可以新增字段，不能修改

快照：

1：安装hdfs仓库插件

bin/elasticsearch-plugin install repository-hdfs

2：构建一个仓库

PUT /\_snapshot/my\_backup

{

"type": "hdfs",

"settings": {

"path": "/back/es/",

"load\_defaults": "true",

"compress": "true",

"uri": "hdfs://192.168.10.160:8020"

}

}

查看仓库信息：

*//查看指定的仓库*

GET /\_snapshot/my\_backup

*//下面当前所有的仓库信息s*

GET /\_snapshot

GET /\_snapshot/\_all

删除一个仓库:

DELETE /\_snapshot/my\_backup

注意删除之后，只是ES里面的引用删除，HDFS上备份的文件是不会删除的

3：构建一个快照

PUT /\_snapshot/my\_backup/snapshot\_1?wait\_for\_completion=true

{

"indices": "index\_1,index\_2",//注意不设置这个属性，默认是备份所有

"ignore\_unavailable": true,

"include\_global\_state": false

}

查询快照的几个方式：

GET /\_snapshot/my\_backup/snapshot\_1 //查询指定快照

GET /\_snapshot/my\_backup/snapshot\_\*,some\_other\_snapshot //支持通配符查询

GET /\_snapshot/my\_backup/\_all //查询所有的快照

删除一个快照：

DELETE /\_snapshot/my\_backup/snapshot\_1

注意删除之后，只是ES里面的引用删除，HDFS上备份的文件是不会删除的

4：恢复快照

POST /\_snapshot/my\_backup/snapshot\_1/\_restore

{

"indices": "index\_1,index\_2", //指定索引恢复，不指定就是所有

"ignore\_unavailable": true,//忽略恢复时异常索引

"include\_global\_state": false,//是否存储全局转态信息,fasle代表有一个或几个失败，不会导致整个任务失败

"rename\_pattern": "index\_(.+)",//是否需要重命名索引

"rename\_replacement": "restored\_index\_$1"//替换后的索引名

}

数据迁移[root@N3 elasticsearch-exporter]# node exporter.js --help

Elasticsearch Exporter - Version 1.4.0

Usage: exporter [options]

Options:

-a <hostname>, --sourceHost <hostname> 迁移源机器地址

-b <hostname>, --targetHost <hostname> 迁移目的机器地址（如果没有设置索引，目的地址需要有别于源地址）

-p <port>, --sourcePort <port> 源机器的ES的端口，9200（一般）

-q <port>, --targetPort <port> 目标机器的ES的端口，9200（一般）

-i <index>, --sourceIndex <index> 源ES待导出的索引，如果该值不设定，整个的数据库都会导出。

-j <index>, --targetIndex <index>目标机器ES的索引，如果源索引设定，该值必须填写。

[root@No3 elasticsearch-exporter]# node exporter.js -a 10.221.110.31-b 100.0.1.130 -p 9200 -q 9200 -i awppx -j awppx

同步最后会显示：

Number of calls: 169

Fetched Entries: 8064 documents

Processed Entries: 8064 documents

Source DB Size: 8064 documents

[root@N3 bin]# cat ./logstash\_output\_mongo/logstash\_es22es.conf

input {

elasticsearch {

hosts => [ "100.200.10.54:9200" ]

index => "doc"

size => 1000

scroll => "5m"

docinfo => true

scan => true

}

}

filter {

json {

source => "message"

remove\_field => ["message"]

}

mutate {

# rename field from 'name' to 'browser\_name'

rename => { "\_id" => "wid" }

}

}

output {

elasticsearch {

hosts => [ "100.20.32.45:9200" ]

document\_type => "docxinfo"

index => "docx"

}

stdout {

codec => "dots"

}

}