# 1、索引的创建于数据添加

**参考资料：**

<https://www.elastic.co/guide/cn/elasticsearch/guide/cn/index.html>

这个是中文版的，虽然这个是es2.x版本的，但是大家可以发现其核心的技术点并没有太多的变化，仍然具备参考价值。

<https://www.elastic.co/guide/en/elasticsearch/reference/6.0/search-aggregations-metrics-avg-aggregation.html>

6.x英文版api资料

Es基础类型：

Text：字符串类型，可以被分析；

Keyword：不能被分析，只可以精确匹配的字符串类型

Date：日期类型，通常配合format使用 比如{“type”:”date”,”format”:”yyyy-MM-dd”}

Long,integer,short…

Boolean

Array：数组类型

Object：一般是json

Ip：ip地址

geo\_point：地理位置 {

“lat”:

“lon”:

}

Put重建索引只会更新数据，不会改变分片副本等配置值

指定分片与副本

PUT /test2

{

"settings": {

"number\_of\_shards": 1,

"number\_of\_replicas": 1

}

}

更新副本数量

PUT /test2/\_settings

{

"number\_of\_replicas": 0

}

添加数据

PUT /test/\_doc/1

{

"name":"kevin",

"age1":"30"

}

结构化索引创建

PUT /test

{

"settings": {

"number\_of\_shards": 1,

"number\_of\_replicas": 1

},

"mappings": {

"\_doc":{

"properties":{

"name":{

"type":"text","analyzer":"ik\_max\_word","search\_analyzer":"ik\_smart"

},

"sname":{

"type":"text","analyzer":"ik\_smart"

},

"enname":{

"type":"text","analyzer":"english"

},

"age":{

"type":"intrger"

}

}

}

}

}

使用put会重建索引，如果只修改数据可以使用post

POST /test/\_doc/1/\_update

{

"doc":{

"name":"kevin1"

}

}

如果post不加\_update会更新所有数据等同于put

POST /test/\_doc/2

{

"age":3

}

加上\_create后只能执行一次，否则会报错

POST /test/\_doc/3/\_create

{

"age":3

}

当post创建数据不指定id时，es会指定一个默认ID，并且可以重复创建

POST /test/\_doc{

"age":3

}

{

"\_index" : "test",

"\_type" : "\_doc",

"\_id" : "PmzHeW8BTfPcbtGGONN4",

"\_score" : 1.0,

"\_source" : {

"age" : 4

}

# 2、数据的查询

## 2.1 kibana查询操作

GET：term为精确匹配，match为模糊查询

### 1 term精确匹配

GET /book/\_search

{

"query": {

"term": {

"bookName": {

"value": "童话故事大全"

}

}

}

Term查询值使用value，match查询值使用query

### 2 短语查询

Match\_phase:查询确切的phase即搜索的值，例如hello world，返回满足如下两个条件的数据，1、match\_phase中所有的term都出现在待查字段中 2、待查字段中所有term都必须和match\_phase具有相同的顺序

match\_phrase\_prefix:可实现即时的短语匹配查询，与match\_phrase查询相同，不同的是它将查询字符串的最后一个词作为前缀使用，即查询时先匹配最后一个词

"match\_phrase\_prefix" : {

"brand" : {

"query": "walker johnnie bl", (1)

"slop": 10，

"max\_expansions": 50

}

}

Slop设置为使搜索的字符串各个词的顺序匹配不那么严格

max\_expansions：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 ，参数 max\_expansions 控制着可以与前缀匹配的词的数量，它会先查找第一个与前缀 bl 匹配的词，然后依次查找搜集与之匹配的词（按字母顺序），直到没有更多可匹配的词或当数量超过 max\_expansions 时结束。

### 3.多字段查询

Multi\_match：多字段查询

GET /book/\_search

{

"query": {

"multi\_match": {

"query": "大自然的旅行故事",

"fields": ["bookName","discription"]

}

}

}

此时为默认best\_fields类型，即取两边分词分数高的那个，多字段查询中如果字段中匹配到多个分词，则此字段分数为多个分词分数之和

字段加权重，即在bookName字段的分数乘10

"fields": ["bookName^10","discription"]

添加tie\_breaker：

"tie\_breaker": 0.3 放在fields下面

表示单一分词在两个字段中的得分取最大值加上其余值的0.3倍

多字段分值相加：

"type": "most\_fields" 即该条数据的总分数为多个字段分数之和

以分词为单位分别在每个字段里面得分取最大相加，非常适用于以词作为权重的系统：

"type": "cross\_fields"，以分词为单位，单个分词的得分取该分词在所有字段中的最大值，然后多个分词的得分之和为该条数据的总得分

### 4.排序、分页、聚合

查询所有

GET /test/\_search

或者

GET /test/\_search

{

"query":{

"match\_all": {}

}

}

分页查询

GET /test/\_search

{

"query":{

"match\_all": {}

},

"from":0,

"size":2

}

带条件查询

GET /test/\_search

{

"query": {

"match": {

"name": "kevin"

}

}

}

排序加聚合查询

GET /test/\_search

{

"query": {

"match": {

"name": "kevin"

}

},

"sort": [

{

"age": {

"order": "desc"

}

}

],

"aggs":{

"group\_by\_age":{

"terms": {

"field": "age",

"size": 4 分组后显示的分组个数

}

}

}

}

### 5.最小匹配查询

GET /book/\_analyze

{

"field": "bookName",

"text": "安徒生的大自然童话故事"

}

GET /book/\_search

{

"query": {

"match": {

"bookName": {

"query": "安徒生的大自然童话故事",

"operator": "or",

"minimum\_should\_match": 2

}

}

}

}

"operator": "or"，or代表搜索的字符串所分的词用于检索式有一个即可，and代表需要包含所有分词

"minimum\_should\_match": 2 必须包含的分词个数，用于控制返回结果数，此时operator需要为or

### 6、analyze、分词器与validate-query

**analyze api**

GET /test/\_analyze

{

"field": "name",

"text": "my name is zhaoyun and i like eating apples and running"

}

从上面的结果中我们可以看出es对齐进行了分词。Es中主要有三种分词：

(1)stander中文的默认分词，这是按照一个个字分分开。此种分词非常占空间会导致倒排索引很大，而且在搜索的时候也会搜出很多不相干的东西。优点就是搜的多。。

(2)English分词器：会提取词干和去掉停用词。假设name不采用英文分词：

PUT /test/\_doc/1

{

"name": "eating apples,

"age": 30

}

然后搜索

GET /test/\_search

{

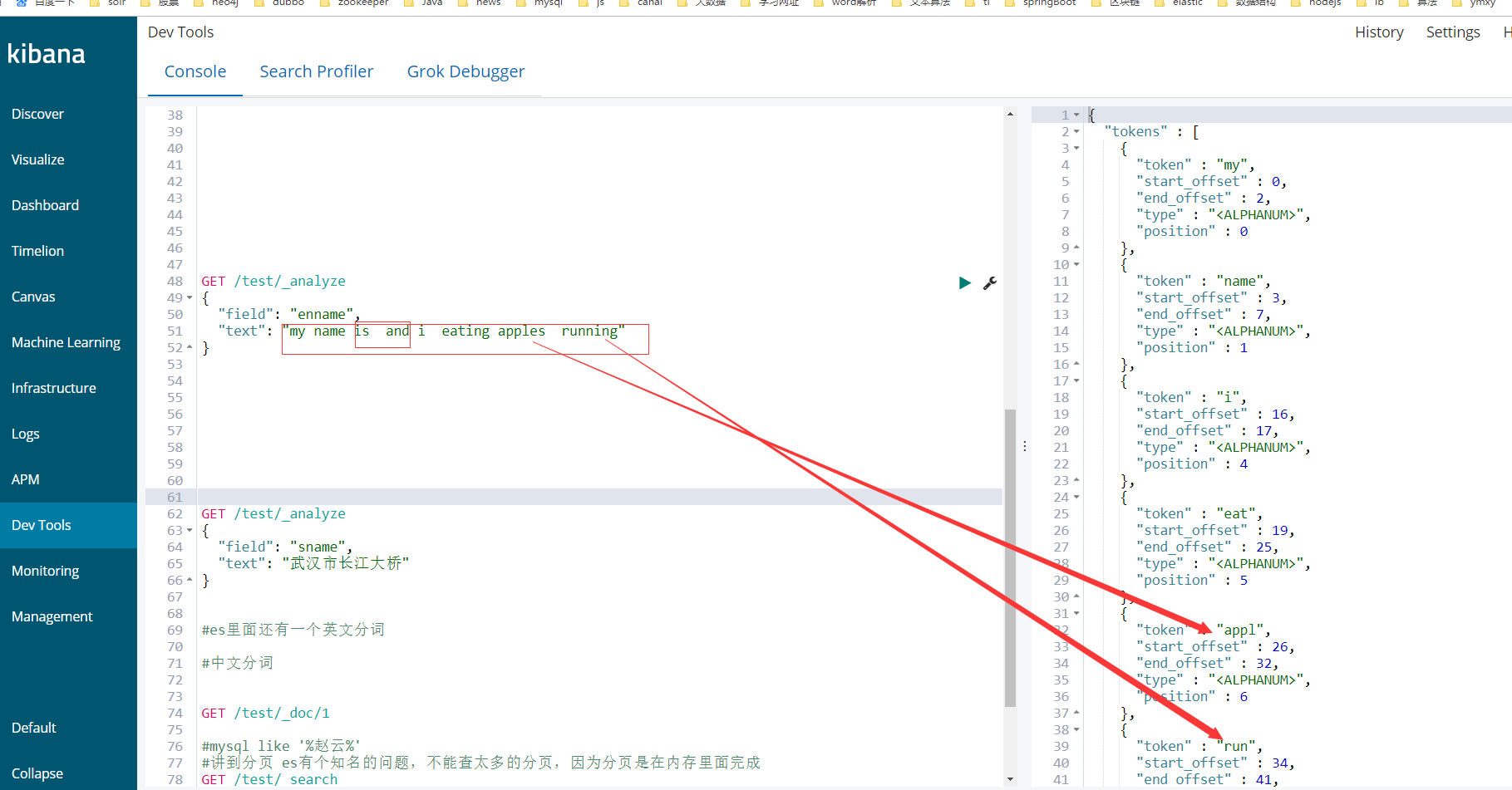
"query":{

"match": {"name":"eat"}

}

}

没搜到结果，我们使用后使用analyze api查看分析处理结果，可以看到没有分出eat，所以搜不到，改成用english分词器做。就可以看到以下的情况



可以看到running和apples被提取成了词干，提取算法其实就是一个映射感兴趣的同学可以自己取看看。而and 和 is被当作停用词去掉了。

（3）IK分词器：他会有两种模式一种smart，一种max\_word. 建索引的时候 用ik\_max\_word，搜索的时候用smart。如果需要详细理解里面的原理的，请看算法课的第23节课。讲的相当详细。

**分词的妙用：**

是不是就意味着stander分词没用了？其实并不是这样的哦

#托底，搜江大桥没有，在建了ik的字段，在建一个一样的stander的字段。如果ik搜不到 就可以搜这一个stander分词的，这样保证会又结果。但是慎用，因为占空间，有些特殊的系统可以使用。

#其实ik也还有一个解决的办法 叫砍词：江大桥 我可以砍掉一个词，我砍掉江 就出来了。砍词的策略可以自定义

#江大桥：电商中。我们系统假设有大桥这个品牌。 我会一个个的是去试一下，比如可以用字符串匹配找出大桥。也有很多系统很粗暴，直接从第一个字开始砍，一直砍到有为止。

#既有英文又有中文的 直接选ik

#如果不用砍词那就要去词库加词，比如加入江大桥就可以了。具体路径是在es的：

/apps/svr/elasticsearch-node1/config/analysis-ik/main.dic

注意集群的话那就要所有的es都需要加的哦

验证查询validate-query：

GET /book/\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "童话故事的大选",

"fields": ["bookName","discription"]

}

}

}

当不加？explain时，返回查询是否合法的信息，当想了解具体原因时需要加上

也可以最直接在query上面加上"explain":true,

GET /book/\_search

{

"explain": true,

"query": {

"multi\_match": {

"query": "大自然的旅行故事",

"fields": ["bookName","discription"],

"type": "cross\_fields"

}

}

}

### 7、queryString

GET /book/\_search

{

"query": {

"query\_string": {

"fields": ["bookName"],

"query": "大自然 AND 旅行"

}

}

}

AND代表包含两个分词，ＯＲ为或，必须大写

### ８、bool过滤器

Bool过滤器由三部分组成：

Must: 所有语句都必须匹配，等价于and

Must\_not: 所有语句都不能匹配，与not等价

Should：至少有一个语句匹配，与or等价

Should与 must\_not 混合使用，should的两个条件满足一个即可，并且必须满足must\_not

GET /test/\_search

{

"query": {

"bool": {

"should": [

{

"term": {

"name": "java"

}

},

{

"term": {

"sname": "develop"

}

}

],

"must\_not": {

"term": {

"name": "java develop"

}

}

}

}

}

在should中嵌套一个bool查询

GET /test/\_search

{

"query" : {

"bool" : {

"should" : [

{ "term" : {"name" : "java"}},

{ "bool" : {

"must" : [

{ "term" : {"sname": "develop"}},

{ "term" : {"age" : 23}}

]

}}

]

}

}

}

当bool查询为should条件时，下方的filter过滤should查出的数据，此部分数据有分数，filter任然会过滤其他不符合should的数据，此部分数据分值为0

GET /test/\_search

{

"query" : {

"bool" : {

"should" : [

{ "term" : {"name" : "java"}},

{ "bool" : {

"must" : [

{ "term" : {"sname": "develop"}},

{ "term" : {"age" : 23}}

]

}}

]

,

"filter": {

"range":{

"age":{

"lte":34,

"gte":20

}

}

}

}

}

}

### 9、Function Score 自定义评分函数

score\_mode:指定如何计算组合得分是（即多个函数之间的分数计算）

multiply： 分数相乘（默认）即每个函数得分相乘

sum：分数相加

avg：分数是平均值

first：具有匹配过滤器的第一个函数被应用

max：使用最高分

min：使用最低分

max\_boost: 将新分数限定为不高于某值，如果大于则赋值为限定值

min\_score：将打分低于此值的数据过滤掉

boost\_mode:代表查询的原有得分（即主查询得分）

multiply：查询分数与功能分数（score\_mode）相乘（默认）

replace：仅使用功能分数，查询分数将被忽略

sum：查询分数与功能分数相加

avg：平均分

max：查询分数与功能分数的最大值

min：查询分数与功能分数的最小值

GET /\_search

{

"query": {

"function\_score": {

"query": { "match\_all": {} },

"boost": "5",

"functions": [

{

"filter": { "match": { "test": "bar" } },

"random\_score": {},

"weight": 23

},

{

"filter": { "match": { "test": "cat" } },

"weight": 42

}

],

"max\_boost": 42,

"score\_mode": "max",

"boost\_mode": "multiply",

"min\_score" : 42

}

}

}

**Es中预定义了一些函数，weight，field\_value\_factor,random\_score,衰减函数 linear、exp、gauss，script\_score**

Weight：权重值，单个函数得出的结果乘以权重值为当前函数的得分

**random\_score：**在相同的分片你片中，如果字段值相同，则得到的分值相同，所以字段最好选择唯一性字段，可以使用\_seq\_no字段，但是该字段在你数据更新时会变动，加入seed种子值可以是每次访问得到相同的分值

"random\_score": {

"seed": 10,

"field":"\_seq\_no"

}

**field\_value\_factor：主查询分值乘以该函数返回值**

factor: 分数乘以相应的倍数

modifier：

none：不使用任何算法公式（默认）

log：常用对数公式

log1p：给字段的值加1，然后使用普通的对数公式，加1使其更加服从高斯分布，避免重复值问题

log2p：给字段的值加2，然后使用普通的对数公式

ln：natural logarithm 自然对数，以e为底的对数

ln1p：给字段的值加1，然后使用自然对数公式

ln2p：给字段的值加2，然后使用自然对数公式

square：平方值

sqrt：开平方

reciprocal：取倒数

missing：如果文档中不包含此字段，则此字段的初始值为missing值

GET /book/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "大自然的旅行故事",

"fields": ["bookName","discription"],

"operator": "or",

"type": "most\_fields"

}

},

"functions": [

{

"field\_value\_factor": {

"field": "commentNum",

"modifier": "log2p",

"factor": 8

“missing”:1

}

}

**script\_score:使用自定义的脚本实现控制评分计算**

script中使用的字段需要为keyword类型，text类型需要将其fielddata=true

如果要忽略主查询得分，可使用boost\_mode：replace

GET /\_search

{

"query": {

"function\_score": {

"query": {

"match": { "message": "elasticsearch" }

},

"script\_score" : {

"script" : {

"params": {

"a": 5,

"b": 1.2

},

"source": "params.a / Math.pow(params.b, doc['name'].value)"

}

}

}

}

}

],

"score\_mode": "sum",

"boost\_mode": "sum"

}

}

}

Decay functions：衰减函数

有三种衰减函数—— linear 、 exp 和 gauss （线性、指数和高斯函数），它们可以操作数值、时间以及经纬度地理坐标点这样的字段。所有三个函数都能接受以下参数

Origin：*中心点* 或字段可能的最佳值，落在原点 origin 上的文档评分 \_score 为满分 1.0

Scale：衰减率，即一个文档从原点 origin 下落时，评分 \_score 改变的速度。（例如，每 £10 欧元或每 100 米）。

Decay：从原点 origin 衰减到 scale 所得的评分 \_score ，默认值为 0.5

Offset：以原点 origin 为中心点，为其设置一个非零的偏移量 offset 覆盖一个范围，而不只是单个原点。在范围 origin +/- offset 内的所有评分 \_score 都是 1.0

Liner:  线性函数是条直线，一旦直线与横轴 0 相交，所有其他值的评分都是 0.0

Exp:指数函数先剧烈衰减然后变缓

Gauss： 高斯函数是钟形的——它的衰减速率是先缓慢，然后变快，最后又放缓