文档：2、ElasticSearch高级语法搜索实战.not...

链接：http://note.youdao.com/noteshare?id=00f521f533f08e8ea9d660e151859fdf&sub=7D084D5E50EC47269F24BB9E01E91744

一. DSL语言高级查询

1. Query DSL概述

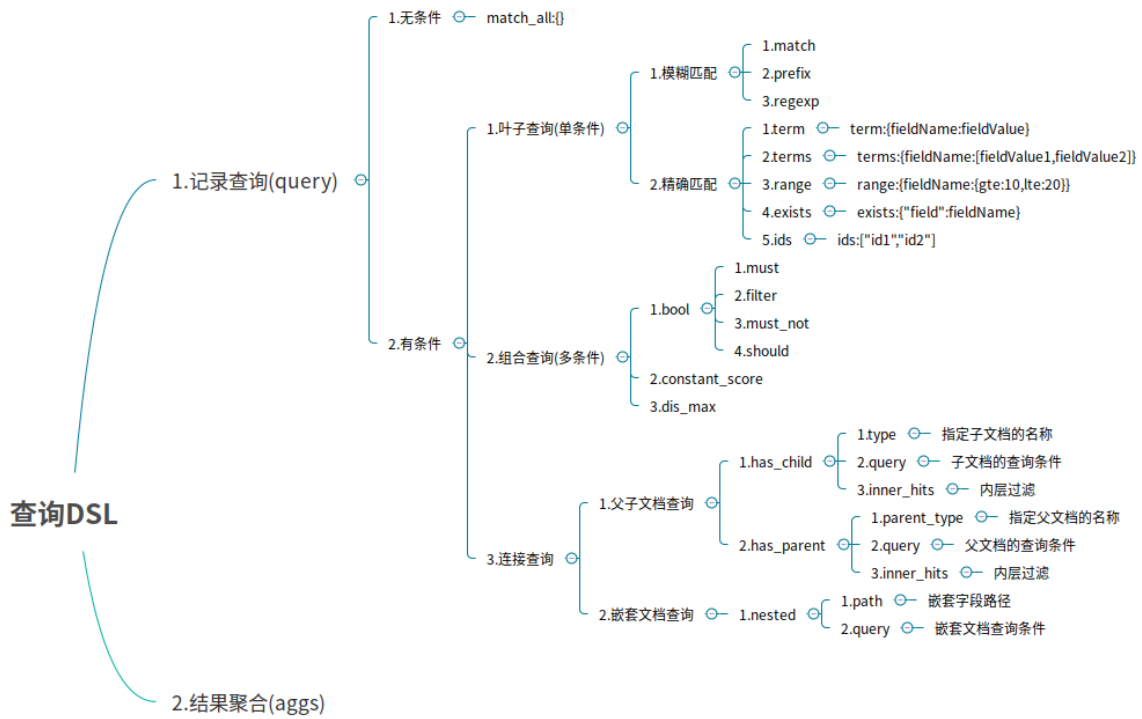
Domain Specific Language

领域专用语言

Elasticsearch provides a ful1 Query DSL based on JSON to define queries

Elasticsearch提供了基于JSON的DSL来定义查询。

DSL由叶子查询子句和复合查询子句两种子句组成。



2. 无查询条件

无查询条件是查询所有，默认是查询所有的，或者使用match_all表示所有

```
1 GET /es_db/_doc/_search
2 {
```

```
"query":{
"match_all":{}
}
```

3. 有查询条件

3.1 叶子条件查询(单字段查询条件)

3.1.1 模糊匹配

模糊匹配主要是针对文本类型的字段，文本类型的字段会对内容进行分词，对查询时，也会对搜索条件进行分词，然后通过倒排索引查找到匹配的数据，模糊匹配主要通过match等参数来实现

- match ： 通过match关键词模糊匹配条件内容
- prefix ： 前缀匹配
- regexp ： 通过正则表达式来匹配数据

match的复杂用法

match条件还支持以下参数：

- query ： 指定匹配的值
- operator ： 匹配条件类型
  - and ： 条件分词后都要匹配
  - or ： 条件分词后有一个匹配即可(默认)
- minmum_should_match ： 指定最小匹配的数量

3.1.2 精确匹配

- term ： 单个条件相等
- terms ： 单个字段属于某个值数组内的值
- range ： 字段属于某个范围内的值
- exists ： 某个字段的值是否存在

- ids ： 通过ID批量查询

3.2 组合条件查询(多条件查询)

组合条件查询是将叶子条件查询语句进行组合而形成的一个完整的查询条件

- bool ： 各条件之间有and,or或not的关系
    - must ： 各个条件都必须满足，即各条件是and的关系
    - should ： 各个条件有一个满足即可，即各条件是or的关系
    - must_not ： 不满足所有条件，即各条件是not的关系
    - filter ： 不计算相关度评分，它不计算_score即相关度评分，效率更高
- constant_score ： 不计算相关度评分

must/filter/shoud/must_not 等的子条件是通过 term/terms/range/ids/exists/match 等叶子条件为参数的

注：以上参数，当只有一个搜索条件时，must等对应的是一个对象，当是多个条件时，对应的是一个数组

3.3 连接查询(多文档合并查询)

- 父子文档查询：parent/child
- 嵌套文档查询: nested

3.4 DSL查询语言中存在两种：查询DSL（query DSL）和过滤DSL（filter DSL）它们两个的区别如下图:

query DSL

在查询上下文中，查询会回答这个问题——“这个文档匹不匹配这个查询，它的相关度高么？”

如何验证匹配很好理解，如何计算相关度呢？ES中索引的数据都会存储一个_score分值，分值越高就代表越匹配。另外关于某个搜索的分值计算还是很复杂的，因此也需要一定的时间。

filter DSL

在过滤器上下文中，查询会回答这个问题——“这个文档匹不匹配？”

答案很简单，是或者不是。它不会去计算任何分值，也不会关心返回的排序问题，因此效率会高一点。

过滤上下文 是在使用filter参数时候的执行环境，比如在bool查询中使用must_not或者filter

另外，经常使用过滤器，ES会自动的缓存过滤器的内容，这对于查询来说，会提高很多性能。

一些过滤的情况：

3.5 Query方式查询:案例

- 根据名称精确查询姓名 term，term查询不会对字段进行分词查询，会采用精确匹配

注意：采用term精确查询，查询字段映射类型属于为keyword.

举例:

```
POST /es_db/_doc/_search
{
"query": {
"term": {
"name": "admin"
}
}
```

```
SQL: select * from student where name = 'admin'
```

- 根据备注信息模糊查询 match，match会根据该字段的分词器，进行分词查询

举例:

```
POST /es_db/_doc/_search
{
"from": 0,
"size": 2,
"query": {
"match": {
"address": "广州"
}
}
```

```
SQL: select * from user where address like '%广州%' limit 0, 2
```

- 多字段模糊匹配查询与精准查询 multi_match

```
POST /es_db/_doc/_search
{
"query":{
"multi_match":{
"query":"张三",
"fields":["address","name"]
}
}
```

```
SQL: select * from student  where name like '%张三%' or address like '%张三%'
```

- 未指定字段条件查询 query_string ， 含 AND 与 OR 条件

```
POST /es_db/_doc/_search
{
"query":{
"query_string":{
"query":"广州 OR 长沙"
}
}
```

- 指定字段条件查询 query_string ， 含 AND 与 OR 条件

```
POST /es_db/_doc/_search
{
"query":{
"query_string":{
"query":"admin OR 长沙",
"fields":["name","address"]
}
}
```

- 范围查询

注：json请求字符串中部分字段的含义

range：范围关键字

gte 大于等于

lte 小于等于

gt 大于

lt 小于

now 当前时间

```
POST /es_db/_doc/_search
{
"query" : {
"range" : {
"age" : {
"gte":25,
"lte":28
}
}
 }
```

```
SQL: select * from user where age between 25 and 28
```

- 分页、输出字段、排序综合查询

```
POST /es_db/_doc/_search
{
"query" : {
"range" : {
"age" : {
"gte":25,
"lte":28
}
}
 },
 "from": 0,
 "size": 2,
 "_source": ["name", "age", "book"],
 "sort": {"age":"desc"}
```

3.6 Filter过滤器方式查询，它的查询不会计算相关性分值，也不会对结果进行排序，因此效率会高一点，查询的结果可以被缓存。

Filter Context 对数据进行过滤

```
POST /es_db/_doc/_search
{
"query" : {
"bool" : {
```

```
"filter" : {
"term":{
"age":25
}
}
}
}
```

总结:

1. match

match：模糊匹配，需要指定字段名，但是输入会进行分词，比如"hello world"会进行拆分为hello和world，然后匹配，如果字段中包含hello或者world，或者都包含的结果都会被查询出来，也就是说match是一个部分匹配的模糊查询。查询条件相对来说比较宽松。

2. term

term: 这种查询和match在有些时候是等价的，比如我们查询单个的词hello，那么会和match查询结果一样，但是如果查询"hello world"，结果就相差很大，因为这个输入不会进行分词，就是说查询的时候，是查询字段分词结果中是否有"hello world"的字样，而不是查询字段中包含"hello world"的字样。当保存数据"hello world"时，elasticsearch会对字段内容进行分词，"hello world"会被分成hello和world，不存在"hello world"，因此这里的查询结果会为空。这也是term查询和match的区别。

3. match_phase

match_phase：会对输入做分词，但是需要结果中也包含所有的分词，而且顺序要求一样。以"hello world"为例，要求结果中必须包含hello和world，而且还要求他们是连着的，顺序也是固定的，hello that world不满足，world hello也不满足条件。

4. query_string

query_string：和match类似，但是match需要指定字段名，query_string是在所有字段中搜索，范围更广泛。

二. 文档映射

1. ES中映射可以分为动态映射和静态映射

动态映射：

在关系数据库中，需要事先创建数据库，然后在该数据库下创建数据表，并创建表字段、类型、长度、主键等，最后才能基于表插入数据。而Elasticsearch中不需要定义Mapping映射（即关系型数据库的表、字段等），在文档写入Elasticsearch时，会根据文档字段自动识别类型，这种机制称之为动态映射。

动态映射规则如下：

JSON数据	自动推测的类型
null	没有字段被添加
true或false	boolean型
小数	float型
数字	long型
日期	date或text
字符串	text
数组	由数组第一个非空值决定
JSON对象	object类型

静态映射：

静态映射是在Elasticsearch中也可以事先定义好映射，包含文档的各字段类型、分词器等，这种方式称之为静态映射。

2 动态映射

2.1 删除原创建的索引

```
DELETE /es_db
```

2.2 创建索引

```
PUT /es_db
```

2.3 创建文档(ES根据数据类型，会自动创建映射)

```
PUT /es_db/_doc/1
{
"name": "Jack",
"sex": 1,
"age": 25,
"book": "java入门至精通",
"address": "广州小蛮腰",
}
```

2.4 获取文档映射

```
GET /es_db/_mapping
```

3 静态映射

3.1 删除原创建的索引

```
DELETE /es_db
```

3.2 创建索引

```
PUT /es_db
```

3.3 设置文档映射

```
PUT /es_db
{
"mappings":{
"properties":{
"name":{"type":"keyword","index":true,"store":true},
"sex":{"type":"integer","index":true,"store":true},
"age":{"type":"integer","index":true,"store":true},
"book":{"type":"text","index":true,"store":true},
"address":{"type":"text","index":true,"store":true}
 }
 }
```

3.4 根据静态映射创建文档

```
PUT /es_db/_doc/1
{
"name": "Jack",
"sex": 1,
"age": 25,
"book": "elasticSearch入门至精通",
"address": "广州车陂"
}
```

3.5 获取文档映射

```
GET /es_db/_mapping
```

三.核心类型（Core datatype）

字符串：string，string类型包含 text 和 keyword。

text：该类型被用来索引长文本，在创建索引前会将这些文本进行分词，转化为词的组合，建立索引；允许es来检索这些词，text类型不能用来排序和聚合。

keyword：该类型不能分词，可以被用来检索过滤、排序和聚合，keyword类型不可用text进行分词模糊检索。

数值型：long、integer、short、byte、double、float

日期型：date

布尔型：boolean

四.keyword 与 text 映射类型的区别

将 book 字段设置为 keyword 映射 （只能精准查询，不能分词查询，能聚合、排序）

```
POST /es_db/_doc/_search
{
"query": {
"term": {
"book": "elasticSearch入门至精通"
}
}
```

将 book 字段设置为 text 映射能模糊查询，能分词查询，不能聚合、排序）

```
POST /es_db/_doc/_search
{
"query": {
"match": {
"book": "elasticSearch入门至精通"
}
}
```

五. 创建静态映射时指定text类型的ik分词器

1. 设置ik分词器的文档映射

先删除之前的es_db

再创建新的es_db

定义ik_smart的映射

```
PUT /es_db
{
"mappings":{
"properties":{
"name":{"type":"keyword","index":true,"store":true},
"sex":{"type":"integer","index":true,"store":true},
"age":{"type":"integer","index":true,"store":true},
"book":{"type":"text","index":true,"store":true,"analyzer":"ik_smart","search_analyzer":"ik_smart"},
"address":{"type":"text","index":true,"store":true}
 }
 }
```

2. 分词查询

```
POST /es_db/_doc/_search
{
"query": {
"match": {"address": "广"}
}
}

POST /es_db/_doc/_search
{
 "query": {
 "match": {"address": "广州"}
 }
 }
```

六. 对已存在的mapping映射进行修改

具体方法

1）如果要推倒现有的映射，你得重新建立一个静态索引

2）然后把之前索引里的数据导入到新的索引里

3）删除原创建的索引

4）为新索引起个别名，为原索引名

```
POST _reindex
{
"source": {
"index": "db_index"
},
"dest": {
"index": "db_index_2"
}
}

DELETE /db_index
PUT /db_index_2/_alias/db_index
```

注意：通过这几个步骤就实现了索引的平滑过渡，并且是零停机

七. 高亮显示

在搜索中，经常需要对搜索关键字做高亮显示，高亮显示也有其常用的参数，在这个案例中做一些常用参数的介绍。

现在搜索name字段中包含“手机”的document。并对“手机”做高亮显示，高亮效果使用html标签，并设定字体为红色。如果name数据过长，则只显示前20个字符。

```
PUT /news_website
{
 "mappings": {

 "properties": {
 "title": {
 "type": "text",
 "analyzer": "ik_max_word"
 },
  "content": {
  "type": "text",
  "analyzer": "ik_max_word"
```

```
 }
 }
 }

}


PUT /news_website
{
 "settings" : {
 "index" : {
 "analysis.analyzer.default.type": "ik_max_word"
 }
 }
}



PUT /news_website/_doc/1
{
 "title": "这是我写的第一篇文章",
 "content": "大家好，这是我写的第一篇文章，特别喜欢这个文章门户网站！！！"
}

GET /news_website/_doc/_search
{
 "query": {
 "match": {
 "title": "文章"
 }
 },
 "highlight": {
 "fields": {
 "title": {}
 }
 }
}

{
 "took" : 458,
 "timed_out" : false,
```

```
 "_shards" : {
 "total" : 1,
 "successful" : 1,
 "skipped" : 0,
 "failed" : 0
 },
 "hits" : {
 "total" : {
 "value" : 1,
 "relation" : "eq"
 },
 "max_score" : 0.2876821,
 "hits" : [
 {
 "_index" : "news_website",
 "_type" : "_doc",
 "_id" : "1",
 "_score" : 0.2876821,
 "_source" : {
 "title" : "我的第一篇文章",
 "content" : "大家好，这是我写的第一篇文章，特别喜欢这个文章门户网站！！！"
 },
 "highlight" : {
 "title" : [
 "我的第一篇<em>文章</em>"
 ]
 }
 }
 ]
 }
}
```

表现，会变成红色，所以说你的指定的field中，如果包含了那个搜索词的话，就会在那个field的文本中，对搜索词进行红色的高亮显示

```
GET /news_website/_doc/_search
{
 "query": {
 "bool": {
 "should": [
 {
```

```
 "match": {
 "title": "文章"
 }
 },
 {
  "match": {
  "content": "文章"
  }
  }
  ]
  }
  },
  "highlight": {
  "fields": {
  "title": {},
  "content": {}
  }
  }
}

highlight中的field，必须跟query中的field一一对齐的


```

八. 手工控制搜索结果精准度

数据准备:

```

PUT /es_db/_doc/1
{
"name": "张三",
"sex": 1,
"age": 25,
"address": "广州天河公园",
"remark": "java developer"
}

 PUT /es_db/_doc/2
 {
 "name": "李四",
 "sex": 1,
```

```
"age": 28,
"address": "广州荔湾大厦",
"remark": "java assistant"
}

PUT /es_db/_doc/3
{
"name": "rod",
"sex": 0,
"age": 26,
"address": "广州白云山公园",
"remark": "php developer"
}

PUT /es_db/_doc/4
{
"name": "admin",
"sex": 0,
"age": 22,
"address": "长沙橘子洲头",
"remark": "python assistant"
}

PUT /es_db/_doc/5
{
"name": "小明",
"sex": 0,
"age": 19,
"address": "长沙岳麓山",
"remark": "java architect assistant"
}
```

1、下述搜索中，如果document中的remark字段包含java或developer词组，都符合搜索条件。

```
GET /es_db/_search
{
"query": {
"match": {
"remark": "java developer"
}
```

```
}
```

如果需要搜索的document中的remark字段，包含java和developer词组，则需要使用下述语法：

```
GET /es_db/_search
{
"query": {
"match": {
"remark": {
"query": "java developer",
"operator": "and"
}
}
}
```

上述语法中，如果将operator的值改为or。则与第一个案例搜索语法效果一致。默认的ES执行搜索的时候，operator就是or。

如果在搜索的结果document中，需要remark字段中包含多个搜索词条中的一定比例，可以使用下述语法实现搜索。其中minimum_should_match可以使用百分比或固定数字。百分比代表query搜索条件中词条百分比，如果无法整除，向下匹配（如，query条件有3个单词，如果使用百分比提供精准度计算，那么是无法除尽的，如果需要至少匹配两个单词，则需要用67%来进行描述。如果使用66%描述，ES则认为匹配一个单词即可。）。固定数字代表query搜索条件中的词条，至少需要匹配多少个。

```
GET /es_db/_search
{
"query": {
"match": {
"remark": {
"query": "java architect assistant",
"minimum_should_match": "68%"
}
}
}
```

如果使用should+bool搜索的话，也可以控制搜索条件的匹配度。具体如下：下述案例代表搜索的document中的remark字段中，必须匹配java、developer、assistant三个词条中的至少2个。

```
GET /es_db/_search
{
"query": {
"bool": {
"should": [
{
"match": {
"remark": "java"
}
},
{
"match": {
"remark": "developer"
}
},
{
"match": {
"remark": "assistant"
}
}
],
"minimum_should_match": 2
}
}
```

2、match 的底层转换

其实在ES中，执行match搜索的时候，ES底层通常都会对搜索条件进行底层转换，来实现最终的搜索结果。如：

```
GET /es_db/_search
{
"query": {
"match": {
"remark": "java developer"
}
}
}

转换后是：
GET /es_db/_search
```

```
{
"query": {
"bool": {
"should": [
{
"term": {
"remark": "java"
}
},
{
"term": {
"remark": {
"value": "developer"
}
}
}
]
}
}
```

```
GET /es_db/_search
{
"query": {
"match": {
"remark": {
"query": "java developer",
"operator": "and"
}
}
}
}

转换后是：
GET /es_db/_search
{
"query": {
"bool": {
"must": [
{
"term": {
"remark": "java"
```

```
}
},
{
"term": {
"remark": {
"value": "developer"
}
}
}
]
}
}
```

```
GET /es_db/_search
{
"query": {
"match": {
"remark": {
"query": "java architect assistant",
"minimum_should_match": "68%"
}
}
}
}
```

转换后为：

```
GET /es_db/_search
{
"query": {
"bool": {
"should": [
{
"term": {
"remark": "java"
}
},
{
"term": {
"remark": "architect"
}
},
```

```
{
"term": {
"remark": "assistant"
}
}
],
"minimum_should_match": 2
}
}
```

建议，如果不怕麻烦，尽量使用转换后的语法执行搜索，效率更高。

如果开发周期短，工作量大，使用简化的写法。

3、boost权重控制

搜索document中remark字段中包含java的数据，如果remark中包含developer或architect，则包含architect的document优先显示。（就是将architect数据匹配时的相关度分数增加）。

一般用于搜索时相关度排序使用。如：电商中的综合排序。将一个商品的销量，广告投放，评价值，库存，单价比较综合排序。在上述的排序元素中，广告投放权重最高，库存权重最低。

```
GET /es_db/_search
{
"query": {
"bool": {
"must": [
{
"match": {
"remark": "java"
}
}
],
"should": [
{
"match": {
"remark": {
"query": "developer",
"boost" : 1
}
}
```

```
},
{
"match": {
"remark": {
"query": "architect",
"boost" : 3
}
}
}
]
}
}
}
```

九、match phrase

短语搜索。就是搜索条件不分词。代表搜索条件不可分割。

如果hello world是一个不可分割的短语，我们可以使用前文学过的短语搜索match phrase来实现。语法如下：

```
GET _search
{
"query": {
"match_phrase": {
"remark": "java assistant"
}
}
```

match phrase原理 -- term position

ES是如何实现match phrase短语搜索的？其实在ES中，使用match phrase做搜索的时候，也是和match类似，首先对搜索条件进行分词-analyze。将搜索条件拆分成hello和world。既然是分词后再搜索，ES是如何实现短语搜索的？

这里涉及到了倒排索引的建立过程。在倒排索引建立的时候，ES会先对document数据进行分词，如：

```
GET _analyze
{
```

```
"text": "hello world, java spark",
"analyzer": "standard"
}
```

分词的结果是：

```
{
"tokens": [
    {
      "token": "hello",
      "start_offset": 0,
      "end_offset": 5,
      "type": "<ALPHANUM>",
      "position": 0
    },
     {
       "token": "world",
       "start_offset": 6,
       "end_offset": 11,
       "type": "<ALPHANUM>",
       "position": 1
     },
     {
       "token": "java",
       "start_offset": 13,
       "end_offset": 17,
       "type": "<ALPHANUM>",
       "position": 2
     },
     {
       "token": "spark",
       "start_offset": 18,
       "end_offset": 23,
       "type": "<ALPHANUM>",
       "position": 3
     }
  ]
```

从上述结果中，可以看到。ES在做分词的时候，除了将数据切分外，还会保留一个position。position代表的是这个词在整个数据中的下标。当ES执行match

phrase搜索的时候，首先将搜索条件hello world分词为hello和world。然后在倒排索引中检索数据，如果hello和world都在某个document的某个field出现时，那么检查这两个匹配到的单词的position是否是连续的，如果是连续的，代表匹配成功，如果是不连续的，则匹配失败。

经验分享

使用match和proximity search实现召回率和精准度平衡。

召回率：召回率就是搜索结果比率，如：索引A中有100个document，搜索时返回多少个document，就是召回率（recall）。

精准度：就是搜索结果的准确率，如：搜索条件为hello java，在搜索结果中尽可能让短语匹配和hello java离的近的结果排序靠前，就是精准度（precision）。

如果在搜索的时候，只使用match phrase语法，会导致召回率底下，因为搜索结果中必须包含短语（包括proximity search）。

如果在搜索的时候，只使用match语法，会导致精准度底下，因为搜索结果排序是根据相关度分数算法计算得到。

那么如果需要在结果中兼顾召回率和精准度的时候，就需要将match和proximity search混合使用，来得到搜索结果。

测试案例：

```
POST /test_a/_doc/3
{
"f" : "hello, java is very good, spark is also very good"
}

POST /test_a/_doc/4
{
"f" : "java and spark, development language "
}

 POST /test_a/_doc/5
 {
 "f" : "Java Spark is a fast and general-purpose cluster computing system. I
t provides high-level APIs in Java, Scala, Python and R, and an optimized engi
ne that supports general execution graphs."
```

```
}

POST /test_a/_doc/6
{
"f" : "java spark and, development language "
}

GET /test_a/_search
{
"query": {
"match": {
"f": "java spark"
}
}
}

GET /test_a/_search
{
"query": {
"bool": {
"must": [
{
"match": {
"f": "java spark"
}
}
],
"should": [
{
"match_phrase": {
"f": {
"query": "java spark",
"slop" : 50
}
}
}
]
}
}
}
```

十、近似匹配

前文都是精确匹配。如doc中有数据java assistant，那么搜索jave是搜索不到数据的。因为jave单词在doc中是不存在的。

如果搜索的语法是：

```
GET _search
{
"query" : {
"match" : {
"name" : "jave"
}
}
```

如果需要的结果是有特殊要求，如：hello world必须是一个完整的短语，不可分割；或document中的field内，包含的hello和world单词，且两个单词之间离的越近，相关度分数越高。那么这种特殊要求的搜索就是近似搜索。包括hell搜索条件在hello world数据中搜索，包括h搜索提示等都数据近似搜索的一部分。

如何上述特殊要求的搜索，使用match搜索语法就无法实现了。

1、前缀搜索 prefix search

使用前缀匹配实现搜索能力。通常针对keyword类型字段，也就是不分词的字段。

语法：

```
GET /test_a/_search
{
"query": {
"prefix": {
"f.keyword": {
"value": "J"
}
}
}
```

注意：针对前缀搜索，是对keyword类型字段而言。而keyword类型字段数据大小写敏感。

前缀搜索效率比较低。前缀搜索不会计算相关度分数。前缀越短，效率越低。如果使用前缀搜索，建议使用长前缀。因为前缀搜索需要扫描完整的索引内容，所以前缀越长，相对效率越高。

2、通配符搜索

ES中也有通配符。但是和java还有数据库不太一样。通配符可以在倒排索引中使用，也可以在keyword类型字段中使用。

常用通配符：

? - 一个任意字符

* - 0~n个任意字符

```
GET /test_a/_search
{
"query": {
"wildcard": {
"f.keyword": {
"value": "?e*o*"
}
}
}
```

性能也很低，也是需要扫描完整的索引。不推荐使用。

3、正则搜索

ES支持正则表达式。可以在倒排索引或keyword类型字段中使用。

常用符号：

[] - 范围，如： [0-9]是0~9的范围数字

. - 一个字符

+ - 前面的表达式可以出现多次。

```
GET /test_a/_search
{
"query": {
"regexp" : {
"f.keyword" : "[A-z].+"
```

```
}
}
```

性能也很低，需要扫描完整索引。

4、搜索推荐

搜索推荐： search as your type， 搜索提示。如：索引中有若干数据以“hello”开头，那么在输入hello的时候，推荐相关信息。（类似百度输入框）

语法：

```
GET /test_a/_search
{
"query": {
"match_phrase_prefix": {
"f": {
"query": "java s",
"slop" : 10,
"max_expansions": 10
}
 }
 }
```

其原理和match phrase类似，是先使用match匹配term数据（java），然后在指定的slop移动次数范围内，前缀匹配（s），max_expansions是用于指定prefix最多匹配多少个term（单词），超过这个数量就不再匹配了。

这种语法的限制是，只有最后一个term会执行前缀搜索。

5、fuzzy模糊搜索技术

搜索的时候，可能搜索条件文本输入错误，如：hello world -> hello word。这种拼写错误还是很常见的。fuzzy技术就是用于解决错误拼写的（在英文中很有效，在中文中几乎无效。）。其中fuzziness代表value的值word可以修改多少个字母来进行拼写错误的纠正（修改字母的数量包含字母变更，增加或减少字母。）。f代表要搜索的字段名称。

```
GET /test_a/_search
{
"query": {
"fuzzy": {
```

```
"f" : {
"value" : "word",
"fuzziness": 2
}
}
}
```