

主讲老师: Fox

课前提醒:

1. 没用过ES的同学建议先从ES专题的第一节课开始学习
2. ES的Query DSL查询语法很多, 如何选择合适的语法, 同学们需要理解以下几点:
 - 需求: 精确值还是全文?
 - 分词器会影响查询结果, 不同的字段可以指定不同的分词器
 - Elasticsearch 默认会以文档的相关度算分进行排序
 - . . .
3. ES: v7.17.3

1 文档: 2. Elasticsearch 高级查询语法Query D...
2 链接: <http://note.youdao.com/noteshare?id=924a9d435d78784455143b1dda4a874a&sub=5877C14CBD3D41B0BE42E3CAB5977248>

ES倒排索引

文档映射Mapping

常用Mapping参数配置

Index Template

Dynamic Template

ES高级查询Query DSL

查询所有match_all

返回指定条数size

分页查询from

深分页查询Scroll

指定字段排序sort

返回指定字段_source

短语查询match_phrase

多字段查询multi_match

query_string

simple_query_string

关键词查询Term

前缀查询prefix

通配符查询wildcard

范围查询range

日期range

多id查询ids

模糊查询fuzzy

高亮highlight

自定义高亮html标签

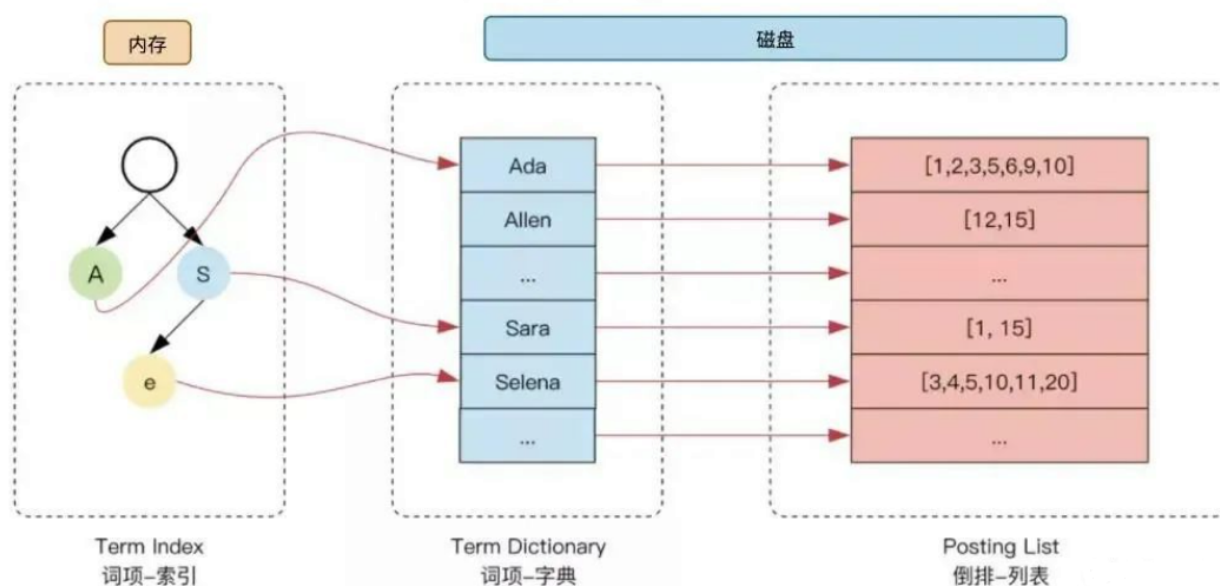
多字段高亮

ES倒排索引

当数据写入 ES 时，数据将会通过 分词 被切分为不同的 term，ES 将 term 与其对应的文档列表建立一种映射关系，这种结构就是 倒排索引。如下图所示：



为了进一步提升索引的效率，ES 在 term 的基础上利用 term 的前缀或者后缀构建了 term index, 用于对 term 本身进行索引，ES 实际的索引结构如下图所示：



这样当我们去搜索某个关键词时，ES 首先根据它的前缀或者后缀迅速缩小关键词的在 term dictionary 中的范围，大大减少了磁盘IO的次数。

- 单词词典 (Term Dictionary)：记录所有文档的单词，记录单词到倒排列表的关联关系
 - 常用字典数据结构：
<https://www.cnblogs.com/LBSer/p/4119841.html>
- 倒排列表(Posting List)-记录了单词对应的文档结合，由倒排索引项组成
- 倒排索引项(Posting):
 - 文档ID
 - 词频TF-该单词在文档中出现的次数，用于相关性评分
 - 位置(Position)-单词在文档中分词的位置。用于短语搜索 (match phrase query)
 - 偏移(Offset)-记录单词的开始结束位置，实现高亮显示

```
#ES的默认分词设置是standard，会单字拆分
POST _analyze
{
  "analyzer": "standard",
  "text": "中华人民共和国"
}

#ik_smart:会做最粗粒度的拆
POST _analyze
{
  "analyzer": "ik_smart",
  "text": "中华人民共和国"
}

#ik_max_word:会将文本做最细粒度的拆分
POST _analyze
{
  "analyzer": "ik_max_word",
  "text": "中华人民共和国"
}
```

```
1 {
2   "tokens" : [
3     {
4       "token" : "中华人民共和国",
5       "start_offset" : 0,
6       "end_offset" : 7,
7       "type" : "CN_WORD",
8       "position" : 0
9     },
10    {
11      "token" : "中华人民",
12      "start_offset" : 0,
13      "end_offset" : 4,
14      "type" : "CN_WORD",
15      "position" : 1
16    },
17    {
18      "token" : "中华",
19      "start_offset" : 0,
20      "end_offset" : 2,
21      "type" : "CN_WORD",
22      "position" : 2
23    },
24  ]
}
```

Elasticsearch 的JSON文档中的每个字段，都有自己的倒排索引。

可以指定对某些字段不做索引：

- 优点：节省存储空间
- 缺点：字段无法被搜索

文档映射Mapping

Mapping类似数据库中的schema的定义，作用如下：

- 定义索引中的字段的名称
- 定义字段的数据类型，例如字符串，数字，布尔等
- 字段，倒排索引的相关配置(Analyzer)

ES中Mapping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。

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。

动态映射（Dynamic Mapping）的机制，使得我们无需手动定义Mappings，Elasticsearch会自动根据文档信息，推算出字段的类型。但是有时候会推算的不对，例如地理位置信息。当类型如果设置不对时，会导致一些功能无法正常运行，例如Range查询

Dynamic Mapping类型自动识别：

JSON类型	Elasticsearch 类型
字符串	<ul style="list-style-type: none">匹配日期格式，设置成 Date配置数字设置为 float或者long，该选项默认关闭设置为Text，并且增加 keyword 子字段
布尔值	boolean
浮点数	float
整数	long
对象	Object
数组	由第一个非空数值的类型所决定
空值	忽略

示例

```
1 #删除原索引
2 DELETE /user
3
4 #创建文档(ES根据数据类型，会自动创建映射)
5 PUT /user/_doc/1
6 {
7   "name": "fox",
8   "age": 32,
9   "address": "长沙麓谷"
10 }
11
12 #获取文档映射
13 GET /user/_mapping
```

```
#动态映射
#删除原索引
DELETE /user

#创建文档(ES根据数据类型,会自动创建映射)
PUT /user/_doc/1
{
  "name": "fox",
  "age": 32,
  "address": "长沙麓谷"
}

#获取文档映射
GET /user/_mapping
```

```
1 {
2   "user" : {
3     "mappings" : {
4       "properties" : {
5         "address" : {
6           "type" : "text",
7           "fields" : {
8             "keyword" : {
9               "type" : "keyword",
10              "ignore_above" : 256
11            }
12          }
13        },
14        "age" : {
15          "type" : "long"
16        },
17        "name" : {
18          "type" : "text",
19          "fields" : {
20            "keyword" : {
21              "type" : "keyword",
22              "ignore_above" : 256
23            }
24          }
25        }
26      }
27    }
28  }
29 }
```

思考：能否后期更改Mapping的字段类型？

两种情况：

- **新增加字段**
 - dynamic设为true时，一旦有新增字段的文档写入，Mapping也同时被更新
 - dynamic设为false，Mapping不会被更新，新增字段的数据无法被索引，但是信息会出现在_source中
 - dynamic设置成strict(严格控制策略)，文档写入失败，抛出异常

	true	false	strict
文档可索引	yes	yes	no
字段可索引	yes	no	no
Mapping被更新	yes	no	no

- **对已有字段，一旦已经有数据写入，就不再支持修改字段定义**
 - Lucene 实现的倒排索引，一旦生成后，就不允许修改
 - 如果希望改变字段类型，可以利用 reindex API，重建索引

原因：

- 如果修改了字段的数据类型，会导致已被索引的数据无法被搜索
- 但是如果是增加新的字段，就不会有这样的影响

测试

```
1 PUT /user
2 {
3   "mappings": {
4     "dynamic": "strict",
5     "properties": {
6       "name": {
7         "type": "text"
8       },
9       "address": {
10        "type": "object",
11        "dynamic": "true"
12      }
13    }
14  }
15 }
16 # 插入文档报错，原因为age为新增字段,会抛出异常
17 PUT /user/_doc/1
18 {
19   "name": "fox",
20   "age": 32,
21   "address": {
22     "province": "湖南",
23     "city": "长沙"
24   }
25 }
```

dynamic设置成strict，新增age字段导致文档插入失败

The screenshot displays a REST client interface with two panels. The left panel shows the request details for a PUT operation on `/user/_doc/1`. The request body is a JSON object: `{ "name": "fox", "age": 32, "address": { "province": "湖南", "city": "长沙" } }`. The right panel shows the error response, which is a JSON object: `{ "error": { "root_cause": [{ "type": "strict_dynamic_mapping_exception", "reason": "mapping set to strict, dynamic introduction of [age] within [_doc] is not allowed" }, { "type": "strict_dynamic_mapping_exception", "reason": "mapping set to strict, dynamic introduction of [age] within [_doc] is not allowed" }], "status": 400 } }`. The error message is highlighted in red in the original image.

修改dynamic后再次插入文档成功

```
1 #修改dynamic
2 PUT /user/_mapping
3 {
4   "dynamic":true
5 }
```

对已有字段的mapping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

```
1
2 PUT /user2
3 {
4   "mappings": {
5     "properties": {
6       "name": {
7         "type": "text"
8       },
9       "address": {
10        "type": "text",
11        "analyzer": "ik_max_word"
12      }
13    }
14  }
15 }
16
17 POST _reindex
18 {
19   "source": {
20     "index": "user"
21   },
22   "dest": {
23     "index": "user2"
24   }
25 }
26
```



```
27 DELETE /user
28
29 PUT /user2/_alias/user
30
31 GET /user
```

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

常用Mapping参数配置

- index: 控制当前字段是否被索引, 默认为true。如果设置为false, 该字段不可被搜索

```
1 DELETE /user
2 PUT /user
3 {
4   "mappings" : {
5     "properties" : {
6       "address" : {
7         "type" : "text",
8         "index": false
9       },
10      "age" : {
11        "type" : "long"
12      },
13      "name" : {
14        "type" : "text"
15      }
16    }
17  }
18 }
19
20 PUT /user/_doc/1
21 {
22   "name": "fox",
23   "address": "广州白云山公园",
24   "age": 30
25 }
26
27 GET /user
28
29 GET /user/_search
```

```

30 {
31   "query": {
32     "match": {
33       "address": "广州"
34     }
35   }
36 }

```

设置为false,address不能被搜索

```

1 {
2   "error": {
3     "root_cause": [
4       {
5         "type": "query_shard_exception",
6         "reason": "failed to create query: Cannot search on field [address] since it is not indexed.",
7         "index_uuid": "LkQZn60NTXmLhUtngGpFwA",
8         "index": "user"
9       }
10    ],
11    "type": "search_phase_execution_exception",
12    "reason": "all shards failed",
13    "phase": "query",
14    "grouped": true,
15    "failed_shards": [
16      {
17        "shard": 0,
18        "index": "user",
19        "node": "6bUwg39USubhnA9gs-2A",
20        "reason": {
21          "type": "query_shard_exception",
22          "reason": "failed to create query: Cannot search on field [address] since it is not indexed.",
23          "index_uuid": "LkQZn60NTXmLhUtngGpFwA",
24          "index": "user",
25          "caused_by": {
26            "type": "illegal_argument_exception",
27            "reason": "Cannot search on field [address] since it is not indexed."
28          }
29        }
30      }
31    ]
32  },
33  "status": 400

```

- 有四种不同基本的index options配置，控制倒排索引记录的内容：
 - docs：记录doc id
 - freqs：记录doc id 和term frequencies（词频）
 - positions：记录doc id / term frequencies / term position
 - offsets：doc id / term frequencies / term position / character offsets

text类型默认记录positions，其他默认为 docs。记录内容越多，占用存储空间越大

```

1 DELETE /user
2 PUT /user
3 {
4   "mappings" : {
5     "properties" : {
6       "address" : {
7         "type" : "text",
8         "index_options": "offsets"
9       },
10      "age" : {

```

```
11  "type" : "long"
12  },
13  "name" : {
14    "type" : "text"
15  }
16  }
17  }
18  }
```

- null_value: 需要对Null值进行搜索，只有keyword类型支持设计Null_Value

```
1  DELETE /user
2  PUT /user
3  {
4    "mappings" : {
5      "properties" : {
6        "address" : {
7          "type" : "keyword",
8          "null_value": "NULL"
9        },
10       "age" : {
11         "type" : "long"
12       },
13       "name" : {
14         "type" : "text"
15       }
16     }
17   }
18 }
19
20 PUT /user/_doc/1
21 {
22   "name":"fox",
23   "age":32,
24   "address":null
25 }
26
27 GET /user/_search
28 {
29   "query": {
30     "match": {
31       "address": "NULL "
```

```

32 }
33 }
34 }
35

```

```

5
7 PUT /user
8 {
9   "mappings" : {
10    "properties" : {
11     "address" : {
12      "type" : "keyword",
13      "null_value": "NULL"
14     },
15     "age" : {
16      "type" : "long"
17     },
18     "name" : {
19      "type" : "text"
20     }
21    }
22   }
23 }
24
25 GET /user/_search
26 {
27   "query": {
28     "match": {
29       "address": "NULL"
30     }
31   }
32 }

```

```

1 {
2   "took" : 2,
3   "timed_out" : false,
4   "_shards" : {
5     "total" : 1,
6     "successful" : 1,
7     "skipped" : 0,
8     "failed" : 0
9   },
10  "hits" : {
11    "total" : {
12      "value" : 1,
13      "relation" : "eq"
14    },
15    "max_score" : 0.6931471,
16    "hits" : [
17      {
18        "_index" : "user",
19        "_type" : "_doc",
20        "_id" : "1",
21        "_score" : 0.6931471,
22        "_source" : {
23          "name" : "fox",
24          "age" : 32,
25          "address" : null
26        }
27      }
28    ]
29  }
30 }

```

- `copy_to`设置：将字段的数值拷贝到目标字段，满足一些特定的搜索需求。
`copy_to`的目标字段不出现在`_source`中。

```

1 # 设置copy_to
2 DELETE /address
3 PUT /address
4 {
5   "mappings" : {
6     "properties" : {
7       "province" : {
8         "type" : "keyword",
9         "copy_to": "full_address"
10      },
11       "city" : {
12         "type" : "text",
13         "copy_to": "full_address"
14      }
15     }
16   },

```

```

17  "settings" : {
18  "index" : {
19  "analysis.analyzer.default.type": "ik_max_word"
20  }
21  }
22  }
23
24  PUT /address/_bulk
25  { "index": { "_id": "1"} }
26  {"province": "湖南","city": "长沙"}
27  { "index": { "_id": "2"} }
28  {"province": "湖南","city": "常德"}
29  { "index": { "_id": "3"} }
30  {"province": "广东","city": "广州"}
31  { "index": { "_id": "4"} }
32  {"province": "湖南","city": "邵阳"}
33
34  GET /address/_search
35  {
36  "query": {
37  "match": {
38  "full_address": {
39  "query": "湖南常德",
40  "operator": "and"
41  }
42  }
43  }
44  }

```

Index Template

Index Templates可以帮助你设定Mappings和Settings，并按照一定的规则，自动匹配到新创建的索引之上

- 模版仅在一个索引被新创建时，才会产生作用。修改模版不会影响已创建的索引
- 你可以设定多个索引模版，这些设置会被“merge”在一起
- 你可以指定“order”的数值，控制“merging”的过程

```

1  PUT /_template/template_default
2  {
3  "index_patterns": ["*"],
4  "order": 0,

```

```

5  "version": 1,
6  "settings": {
7    "number_of_shards": 1,
8    "number_of_replicas": 1
9  }
10 }
11
12
13 PUT /_template/template_test
14 {
15   "index_patterns": ["test*"],
16   "order": 1,
17   "settings": {
18     "number_of_shards": 2,
19     "number_of_replicas": 1
20   },
21   "mappings": {
22     "date_detection": false,
23     "numeric_detection": true
24   }
25 }

```

Index Template的工作方式

当一个索引被新创建时：

- 应用Elasticsearch 默认的settings 和mappings
- 应用order数值低的Index Template 中的设定
- 应用order高的 Index Template 中的设定，之前的设定会被覆盖
- 应用创建索引时，用户所指定的Settings和 Mappings，并覆盖之前模版中的设定

```

1  #查看template信息
2  GET /_template/template_default
3  GET /_template/temp*
4
5  PUT /testtemplate/_doc/1
6  {
7    "orderNo": 1,
8    "createDate": "2022/01/01"
9  }
10 GET /testtemplate/_mapping

```

```
11 GET /testtemplate/_settings
12
13 PUT /testmy
14 {
15   "mappings": {
16     "date_detection": true
17   }
18 }
19
20 PUT /testmy/_doc/1
21 {
22   "orderNo": 1,
23   "createDate": "2022/01/01"
24 }
25
26 GET /testmy/_mapping
```

Dynamic Template

Dynamic Template定义在某个索引的Mapping中。

```
1 #Dynaminc Mapping 根据类型和字段名
2 DELETE my_index
3 PUT my_index/_doc/1
4 {
5   "firstName": "Ruan",
6   "isVIP": "true"
7 }
8
9 GET my_index/_mapping
10 DELETE my_index
11 PUT my_index
12 {
13   "mappings": {
14     "dynamic_templates": [
15       {
16         "strings_as_boolean": {
17           "match_mapping_type": "string",
18           "match": "is*",
19           "mapping": {
20             "type": "boolean"
```

```
21  }
22  }
23  },
24  {
25    "strings_as_keywords": {
26      "match_mapping_type": "string",
27      "mapping": {
28        "type": "keyword"
29      }
30    }
31  }
32  ]
33  }
34  }
35
36  #结合路径
37  PUT /my_test_index
38  {
39    "mappings": {
40      "dynamic_templates": [
41        {
42          "full_name":{
43            "path_match": "name.*",
44            "path_unmatch": "/*.middle",
45            "mapping":{
46              "type": "text",
47              "copy_to": "full_name"
48            }
49          }
50        }
51      ]
52    }
53  }
54
55  PUT /my_test_index/_doc/1
56  {
57    "name":{
58      "first": "John",
59      "middle": "Winston",
60      "last": "Lennon"
```



```
61  }
62  }
63
64
65  GET /my_test_index/_search
66  {
67    "query": {
68      "match": {
69        "full_name": "John"
70      }
71    }
72  }
```

ES高级查询Query DSL

ES中提供了一种强大的检索数据方式,这种检索方式称之为Query DSL (Domain Specified Language) , Query DSL是利用Rest API传递JSON格式的请求体(RequestBody)数据与ES进行交互, 这种方式的丰富查询语法让ES检索变得更强大, 更简洁。

<https://www.elastic.co/guide/en/elasticsearch/reference/7.17/query-dsl.html>

语法:

```
1  GET /es_db/_doc/_search {json请求体数据}
2  可以简化为下面写法
3  GET /es_db/_search {json请求体数据}
```

示例

```
1  #无条件查询, 默认返回10条数据
2  GET /es_db/_search
3  {
4    "query":{
5      "match_all":{}
6    }
7  }
```

```

1  # [types removal] specifying types in search requests is deprecated
2  {
3    "took" : 1,
4    "timed_out" : false,
5    "_shards" : {
6      "total" : 1,
7      "successful" : 1,
8      "skipped" : 0,
9      "failed" : 0
10   },
11   "hits" : {
12     "total" : {
13       "value" : 13,
14       "relation" : "eq"
15     },
16     "max_score" : 1.0,
17     "hits" : [
18       {
19         "_index" : "es_db",
20         "_type" : "_doc",
21         "_id" : "2",
22         "_score" : 1.0,
23         "_source" : {
24           "name" : "李四",
25           "sex" : 1,
26           "age" : 28,
27           "address" : "广州荔湾大厦",
28           "remark" : "java assistant"
29         }
30       }
31     ]
32   }
33 }

```

took:花费的时间

total.value: 符合条件的总文档

hits: 结果集, 默认前10个文档

_index:索引名
_id: 文档的id
_score: 相关度评分
_source:文档原生信息

示例数据

```

1  #指定ik分词器
2  PUT /es_db
3  {
4    "settings" : {
5      "index" : {
6        "analysis.analyzer.default.type": "ik_max_word"
7      }
8    }
9  }
10
11 # 创建文档,指定id
12 PUT /es_db/_doc/1
13 {
14   "name": "张三",
15   "sex": 1,

```

```
16  "age": 25,
17  "address": "广州天河公园",
18  "remark": "java developer"
19  }
20  PUT /es_db/_doc/2
21  {
22  "name": "李四",
23  "sex": 1,
24  "age": 28,
25  "address": "广州荔湾大厦",
26  "remark": "java assistant"
27  }
28
29  PUT /es_db/_doc/3
30  {
31  "name": "王五",
32  "sex": 0,
33  "age": 26,
34  "address": "广州白云山公园",
35  "remark": "php developer"
36  }
37
38  PUT /es_db/_doc/4
39  {
40  "name": "赵六",
41  "sex": 0,
42  "age": 22,
43  "address": "长沙橘子洲",
44  "remark": "python assistant"
45  }
46
47  PUT /es_db/_doc/5
48  {
49  "name": "张龙",
50  "sex": 0,
51  "age": 19,
52  "address": "长沙麓谷企业广场",
53  "remark": "java architect assistant"
54  }
55
```

```
56 PUT /es_db/_doc/6
57 {
58   "name": "赵虎",
59   "sex": 1,
60   "age": 32,
61   "address": "长沙麓谷兴工国际产业园",
62   "remark": "java architect"
63 }
```

查询所有match_all

使用match_all，默认只会返回10条数据。

原因：_search查询默认采用的是分页查询，每页记录数size的默认值为10。如果想显示更多数据，指定size

```
1 GET /es_db/_search
2 等同于
3 GET /es_db/_search
4 {
5   "query":{
6     "match_all":{}
7   }
8 }
```

返回指定条数size

size 关键字: 指定查询结果中返回指定条数。默认返回值10条

```
1 GET /es_db/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "size": 100
7 }
8
```

思考：size可以无限增加吗？

测试

```
1 GET /es_db/_search
2 {
3   "query": {
```

```

4  "match_all": {}
5  },
6  "size": 20000
7  }

```

出现异常：

The screenshot shows a REST client interface. On the left, a GET request to `/es_db/_search` is shown with a query object containing `"match_all": {}` and a `"size": 20000` parameter. On the right, the response is an error object. The `"error"` field contains a `"root_cause"` array with one element: `{ "type": "illegal_argument_exception", "reason": "Result window is too large, from + size must be less than or equal to: [10000] but was [20000]. See the scroll api for a more efficient way to request large data sets. This limit can be set by changing the [index.max_result_window] index level setting." }`. The `"type"` is `"search_phase_execution_exception"`, the `"reason"` is `"all shards failed"`, and the `"phase"` is `"query"`. The `"failed_shards"` array contains one entry for shard 0 of index `"es_db"`.

异常原因：

- 1、查询结果的窗口太大，`from + size`的结果必须小于或等于10000，而当前查询结果的窗口为20000。
- 2、可以采用scroll api更高效的请求大量数据集。
- 3、查询结果的窗口的限制可以通过参数`index.max_result_window`进行设置。

```

1  PUT /es_db/_settings
2  {
3    "index.max_result_window" : "20000"
4  }
5  #修改现有所有的索引，但新增的索引，还是默认的10000
6  PUT /_all/_settings
7  {
8    "index.max_result_window" : "20000"
9  }
10
11 #查看所有索引中的index.max_result_window值
12 GET /_all/_settings/index.max_result_window

```

注意：参数`index.max_result_window`主要用来限制单次查询满足查询条件的结果窗口的大小，窗口大小由`from + size`共同决定。不能简单理解成查询返回给调用方的数据量。这样做主要是为了限制内存的消耗。

比如：`from`为1000000，`size`为10，逻辑意义是从满足条件的数据中取1000000到（1000000 + 10）的记录。这时ES一定要先将（1000000 + 10）的记录（即`result_window`）加载到内存中，再进行分页取值的操作。尽管最后我们只取了10条数据返回给客户端，但ES进程执行查询操作的过程中确需要将（1000000 + 10）的记录都加载到内存中，可想而知对内存的消耗有多大。这也是ES中不推荐采用（`from + size`）方式进行深度分页的原因。

同理，from为0，size为1000000时，ES进程执行查询操作的过程中确需要将1000000 条记录都加载到内存中再返回给调用方，也会对ES内存造成很大压力。

分页查询form

from 关键字: 用来指定起始返回位置，和size关键字连用可实现分页效果

```
1 GET /es_db/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "size": 5,
7   "from": 0
8 }
```

深分页查询Scroll

改动index.max_result_window参数值的大小，只能解决一时的问题，当索引的数据量持续增长时，在查询全量数据时还是会出现问题。而且会增加ES服务器内存大结果集消耗完的风险。最佳实践还是根据异常提示中的采用scroll api更高效请求大量数据集。

```
1 #查询命令中新增scroll=1m,说明采用游标查询，保持游标查询窗口一分钟。
2 #这里由于测试数据量不够，所以size值设置为2。
3 #实际使用中为了减少游标查询的次数，可以将值适当增大，比如设置为1000。
4 GET /es_db/_search?scroll=1m
5 {
6   "query": { "match_all": {} },
7   "size": 2
8 }
```

查询结果：

除了返回前2条记录，还返回了一个游标ID值_scroll_id。

```
GET /es_db/_search?scroll=1m
{
  "query": { "match_all": {} },
  "size": 2
}

1- {
2   "scroll_id": "FGluY2x1ZGVyY29udGV4dF91dWlkDXF1ZXJ5QW5kRmV0Y2g8FmNwcVdjb1RxcUzVhZXlicG9HeU02bWcAAAAAABmzRY2YlV3Z0o5SVVNTdWJ0bkE5Z3MtXzJB",
3   "took": 0,
4   "timed_out": false,
5   "shards": {
6     "total": 1,
7     "successful": 1,
8     "skipped": 0,
9     "failed": 0
10  },
11  "hits": { }
12 }
```

采用游标id查询：

```
1 # scroll_id 的值就是上一个请求中返回的 _scroll_id 的值
2 GET /_search/scroll
3 {
4   "scroll": "1m",
```

```

5   "scroll_id" : "FGluY2x1ZGVfY29udGV4dF91dWlkDXF1ZXJ5QW5kRmV0Y2gBFmNwcVdjb
1RxUzVhZXlicG9HeU02bWcAAAAAABmzRY2Y1V3Z0o5VVNTdWJobkE5Z3MtXzJB"
6 }

```



多次根据scroll_id游标查询，直到没有数据返回则结束查询。采用游标查询索引全量数据，更安全高效，限制了单次对内存的消耗。

指定字段排序sort

注意：会让得分失效

```

1 GET /es_db/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "sort": [
7     {
8       "age": "desc"
9     }
10  ]
11 }
12
13 #排序，分页
14 GET /es_db/_search
15 {
16   "query": {
17     "match_all": {}
18   },
19   "sort": [
20     {
21       "age": "desc"
22     }
23 ],

```

```
24   "from": 10,  
25   "size": 5  
26 }
```

返回指定字段_source

_source 关键字: 是一个数组,在数组中用来指定展示那些字段

```
1 GET /es_db/_search  
2 {  
3   "query": {  
4     "match_all": {}  
5   },  
6   "_source": ["name","address"]  
7 }
```

match

match在匹配时会对所查找的关键词进行分词, 然后按分词匹配查找

match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)
- minnum_should_match : 最低匹配度, 即条件在倒排索引中最低的匹配度

```
1 #模糊匹配 match 分词后or的效果  
2 GET /es_db/_search  
3 {  
4   "query": {  
5     "match": {  
6       "address": "广州白云山公园"  
7     }  
8   }  
9 }  
10  
11 # 分词后 and的效果  
12 GET /es_db/_search  
13 {  
14   "query": {
```



```
15  "match": {
16    "address": {
17      "query": "广州白云山公园",
18      "operator": "AND"
19    }
20  }
21 }
22 }
23
```

在match中的应用：当operator参数设置为or时，`minnum_should_match`参数用来控制匹配的分词的最少数量。

```
1  # 最少匹配广州，公园两个词
2  GET /es_db/_search
3  {
4    "query": {
5      "match": {
6        "address": {
7          "query": "广州公园",
8          "minimum_should_match": 2
9        }
10     }
11  }
12 }
```

短语查询match_phrase

match_phrase查询分析文本并根据分析的文本创建一个短语查询。`match_phrase` 会将检索关键词分词。match_phrase的分词结果必须在被检索字段的分词中都包含，而且顺序必须相同，而且默认必须都是连续的。

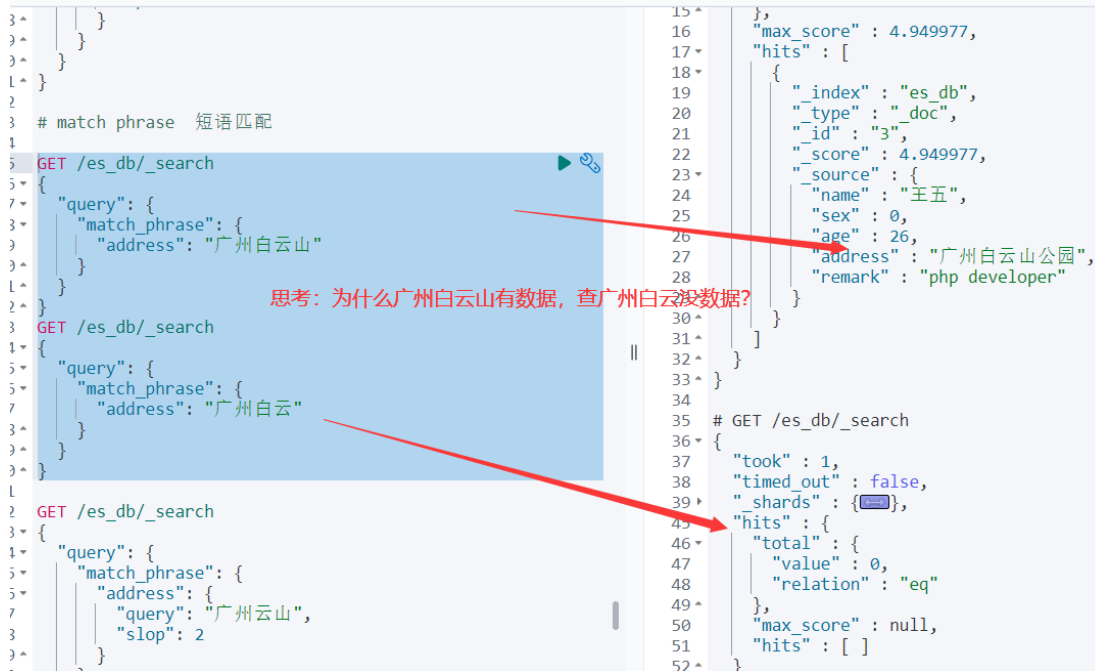
```
1  GET /es_db/_search
2  {
3    "query": {
4      "match_phrase": {
5        "address": "广州白云山"
6      }
7    }
8  }
```

```

9 GET /es_db/_search
10 {
11   "query": {
12     "match_phrase": {
13       "address": "广州白云"
14     }
15   }
16 }

```

思考：为什么查询广州白云山有数据，广州白云没有数据？



```

3 * | }
3 * | }
3 * | }
1 * }
2
3 # match phrase 短语匹配
1
5 GET /es_db/_search
5 {
7   "query": {
9     "match_phrase": {
10      "address": "广州白云山"
11    }
12  }
13 }
2 * }
3
3 GET /es_db/_search
1 {
3   "query": {
5     "match_phrase": {
7       "address": "广州白云"
8     }
9   }
10 }
1
2 GET /es_db/_search
3 {
5   "query": {
7     "match_phrase": {
9       "address": {
10        "query": "广州云山",
11        "slop": 2
12      }
13     }
14   }
15 }
16
15 * },
16   "max_score" : 4.949977,
17   "hits" : [
18     {
19       "_index" : "es_db",
20       "_type" : "doc",
21       "_id" : "3",
22       "_score" : 4.949977,
23       "_source" : {
24         "name" : "王五",
25         "sex" : 0,
26         "age" : 26,
27         "address" : "广州白云山公园",
28         "remark" : "php developer"
29       }
30     }
31   ]
32 }
33
34
35 # GET /es_db/_search
36 {
37   "took" : 1,
38   "timed_out" : false,
39   "_shards" : {
40     "total" : 1,
41     "successful" : 1,
42     "skipped" : 0,
43     "failed" : 0
44   },
45   "hits" : {
46     "total" : {
47       "value" : 0,
48       "relation" : "eq"
49     },
50     "max_score" : null,
51     "hits" : [ ]
52   }
53 }

```

分析原因：

先查看广州白云山公园分词结果，可以知道广州和白云不是相邻的词条，中间会隔一个白云山，而match_phrase匹配的是相邻的词条，所以查询广州白云山有结果，但查询广州白云没有结果。

```

1 POST _analyze
2 {
3   "analyzer": "ik_max_word",
4   "text": "广州白云山"
5 }
6 #结果
7 {
8   "tokens" : [
9     {
10      "token" : "广州",
11      "start_offset" : 0,
12      "end_offset" : 2,

```

```
13  "type" : "CN_WORD",
14  "position" : 0
15  },
16  {
17    "token" : "白云山",
18    "start_offset" : 2,
19    "end_offset" : 5,
20    "type" : "CN_WORD",
21    "position" : 1
22  },
23  {
24    "token" : "白云",
25    "start_offset" : 2,
26    "end_offset" : 4,
27    "type" : "CN_WORD",
28    "position" : 2
29  },
30  {
31    "token" : "云山",
32    "start_offset" : 3,
33    "end_offset" : 5,
34    "type" : "CN_WORD",
35    "position" : 3
36  }
37 ]
38 }
```

如何解决词条间隔的问题？可以借助slop参数，slop参数告诉match_phrase查询词条能够相隔多远时仍然将文档视为匹配。

```
1  #广州云山分词后相隔为2，可以匹配到结果
2  GET /es_db/_search
3  {
4    "query": {
5      "match_phrase": {
6        "address": {
7          "query": "广州云山",
8          "slop": 2
9        }
10     }
11  }
12 }
```

多字段查询multi_match

可以根据字段类型，决定是否使用分词查询，得分最高的在前面

```
1 GET /es_db/_search
2 {
3   "query": {
4     "multi_match": {
5       "query": "长沙张龙",
6       "fields": [
7         "address",
8         "name"
9       ]
10    }
11  }
12 }
```

注意：字段类型分词,将查询条件分词之后进行查询，如果该字段不分词就会将查询条件作为整体进行查询。

query_string

允许我们在单个查询字符串中指定AND | OR | NOT条件，同时也和 multi_match query 一样，支持多字段搜索。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。

注意: 查询字段分词就将查询条件分词查询，查询字段不分词将查询条件不分词查询

- 未指定字段查询

```
1 GET /es_db/_search
2 {
3   "query": {
4     "query_string": {
5       "query": "张三 OR 橘子洲"
6     }
7   }
8 }
```

- 指定单个字段查询

```
1 #Query String
```

```

2 GET /es_db/_search
3 {
4   "query": {
5     "query_string": {
6       "default_field": "address",
7       "query": "白云山 OR 橘子洲"
8     }
9   }
10 }

```

- **指定多个字段查询**

```

1 GET /es_db/_search
2 {
3   "query": {
4     "query_string": {
5       "fields": ["name", "address"],
6       "query": "张三 OR (广州 AND 王五)"
7     }
8   }
9 }
10

```

simple_query_string

类似Query String，但是会忽略错误的语法,同时只支持部分查询语法，不支持AND OR NOT，会当作字符串处理。支持部分逻辑：

- + 替代AND
- | 替代OR
- - 替代NOT

```

1 #simple_query_string 默认的operator是OR
2 GET /es_db/_search
3 {
4   "query": {
5     "simple_query_string": {
6       "fields": ["name", "address"],
7       "query": "广州公园",
8       "default_operator": "AND"
9     }
10   }
11 }

```

```

12
13 GET /es_db/_search
14 {
15   "query": {
16     "simple_query_string": {
17       "fields": ["name","address"],
18       "query": "广州 + 公园"
19     }
20   }
21 }

```

关键词查询Term

Term用来使用关键词查询(精确匹配),还可以用来查询没有被进行分词的数据类型。Term是表达语意的最小单位,搜索和利用统计语言模型进行自然语言处理都需要处理Term。

match在匹配时会对所查找的关键词进行分词,然后按分词匹配查找,而term会直接对关键词进行查找。一般模糊查找的时候,多用match,而精确查找时可以使用term。

- ES中默认使用分词器为标准分词器(StandardAnalyzer),标准分词器对于英文单词分词,对于中文单字分词。
- 在ES的Mapping Type 中 keyword , date ,integer, long , double , boolean or ip 这些类型不分词,只有text类型分词。

```

1 #关键字查询 term
2 # 思考: 查询广州白云是否有数据,为什么?
3 GET /es_db/_search
4 {
5   "query":{
6     "term": {
7       "address": {
8         "value": "广州白云"
9       }
10    }
11  }
12 }
13
14 # 采用term精确查询, 查询字段映射类型为keyword
15 GET /es_db/_search
16 {
17   "query":{

```

```
18  "term": {
19    "address.keyword": {
20      "value": "广州白云山公园"
21    }
22  }
23 }
24 }
```

在ES中，Term查询，对输入不做分词。会将输入作为一个整体，在倒排索引中查找准确的词项，并且使用相关度算分公式为每个包含该词项的文档进行相关度算分。

```
1  PUT /product/_bulk
2  {"index":{"_id":1}}
3  {"productId":"xxx123","productName":"iPhone"}
4  {"index":{"_id":2}}
5  {"productId":"xxx111","productName":"iPad"}
6
7  # 思考： 查询iPhone可以查到数据吗？
8  GET /product/_search
9  {
10   "query":{
11     "term": {
12       "productName": {
13         "value": "iPhone"
14       }
15     }
16   }
17 }
18
19 GET /product/_analyze
20 {
21   "analyzer":"standard",
22   "text":"iPhone"
23 }
24
25 # 对于英文，可以考虑建立索引时忽略大小写
26 PUT /product
27 {
28   "settings": {
29     "analysis": {
30       "normalizer": {
```

```

31   "es_normalizer": {
32     "filter": [
33       "lowercase",
34       "asciifolding"
35     ],
36     "type": "custom"
37   }
38 }
39 }
40 },
41 "mappings": {
42   "properties": {
43     "productId": {
44       "type": "text"
45     },
46     "productName": {
47       "type": "keyword",
48       "normalizer": "es_normalizer",
49       "index": "true"
50     }
51   }
52 }
53 }

```

可以通过 Constant Score 将查询转换成一个 Filtering，避免算分，并利用缓存，提高性能。

- 将Query 转成 Filter，忽略TF-IDF计算，避免相关性算分的开销
- Filter可以有效利用缓存

```

1 GET /es_db/_search
2 {
3   "query": {
4     "constant_score": {
5       "filter": {
6         "term": {
7           "address.keyword": "广州白云山公园"
8         }
9       }
10    }
11  }

```


ES中的结构化搜索

结构化搜索(Structured search)是指对结构化数据的搜索。

结构化数据：

- 日期，布尔类型和数字都是结构化的
- 文本也可以是结构化的。
 - 如彩色笔可以有离散的颜色集合：红(red)、绿(green、蓝(blue)
 - 一个博客可能被标记了标签，例如，分布式(distributed)和搜索(search)
 - 电商网站上的商品都有UPC(通用产品码Universal Product Code)或其他的唯一

标识，它们都需要遵从严格规定的、结构化的格式。

应用场景：对bool，日期，数字，结构化的文本可以利用term做精确匹配

```
1 GET /es_db/_search
2 {
3   "query": {
4     "term": {
5       "age": {
6         "value": 28
7       }
8     }
9   }
10 }
```

term处理多值字段，term查询是包含，不是等于

```
1 POST /employee/_bulk
2 {"index":{"_id":1}}
3 {"name":"小明","interest":["跑步","篮球"]}
4 {"index":{"_id":2}}
5 {"name":"小红","interest":["跳舞","画画"]}
6 {"index":{"_id":3}}
7 {"name":"小丽","interest":["跳舞","唱歌","跑步"]}
8
9 POST /employee/_search
```

```
10 {
11   "query": {
12     "term": {
13       "interest.keyword": {
14         "value": "跑步"
15       }
16     }
17   }
18 }
```

前缀查询prefix

它会对分词后的term进行前缀搜索。

- 它不会分析要搜索字符串，传入的前缀就是想要查找的前缀
- 默认状态下，前缀查询不做相关度分数计算，它只是将所有匹配的文档返回，然后赋予所有相关分数值为1。它的行为更像是一个过滤器而不是查询。两者实际的区别就是过滤器是可以被缓存的，而前缀查询不行。

prefix的原理：需要遍历所有倒排索引，并比较每个term是否已所指定的前缀开头。

```
1 GET /es_db/_search
2 {
3   "query": {
4     "prefix": {
5       "address": {
6         "value": "广州"
7       }
8     }
9   }
10 }
```

通配符查询wildcard

通配符查询：工作原理和prefix相同，只不过它不是只比较开头，它能支持更为复杂的匹配模式。

```
1 GET /es_db/_search
2 {
3   "query": {
4     "wildcard": {
5       "address": {
6         "value": "*白*"
7       }
8     }
9   }
10 }
```

```
7  }
8  }
9  }
10 }
```

范围查询range

- 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

```
1 POST /es_db/_search
2 {
3   "query": {
4     "range": {
5       "age": {
6         "gte": 25,
7         "lte": 28
8       }
9     }
10  }
11 }
```

日期range

```
1 DELETE /product
2 POST /product/_bulk
3 {"index":{"_id":1}}
4 {"price":100,"date":"2021-01-01","productId":"XHDK-1293"}
5 {"index":{"_id":2}}
6 {"price":200,"date":"2022-01-01","productId":"KDKE-5421"}
7
8 GET /product/_mapping
9
10 GET /product/_search
11 {
12   "query": {
```

```
13  "range": {
14  "date": {
15  "gte": "now-2y"
16  }
17  }
18  }
19  }
```

多id查询ids

ids 关键字：值为数组类型,用来根据一组id获取多个对应的文档

```
1  GET /es_db/_search
2  {
3  "query": {
4  "ids": {
5  "values": [1,2]
6  }
7  }
8  }
```

模糊查询fuzzy

在实际的搜索中，我们有时候会打错字，从而导致搜索不到。在Elasticsearch中，我们可以使用fuzziness属性来进行模糊查询，从而达到搜索有错别字的情形。

fuzzy 查询会用到两个很重要的参数，fuzziness, prefix_length

- fuzziness：表示输入的关键字通过几次操作可以转变成为ES库里面的对应field的字段
 - 操作是指：新增一个字符，删除一个字符，修改一个字符，每次操作可以记做编辑距离为1，
 - 如中文集团到中威集团编辑距离就是1，只需要修改一个字符；
 - 该参数默认值为0，即不开启模糊查询。
 - 如果fuzziness值在这里设置成2，会把编辑距离为2的东东集团也查出来。
- prefix_length：表示限制输入关键字和ES对应查询field的内容开头的第n个字符必须完全匹配，不允许错别字匹配
 - 如这里等于1，则表示开头的字必须匹配，不匹配则不返回

- 默认值也是0
- 加大prefix_length的值可以提高效率和准确率。

```
1 GET /es_db/_search
2 {
3   "query": {
4     "fuzzy": {
5       "address": {
6         "value": "白运山",
7         "fuzziness": 1
8       }
9     }
10  }
11 }
12
13 GET /es_db/_search
14 {
15   "query": {
16     "match": {
17       "address": {
18         "query": "广洲",
19         "fuzziness": 1
20       }
21     }
22  }
23 }
24
```

注意: fuzzy 模糊查询 最大模糊错误 必须在0-2之间

- 搜索关键词长度为 2，不允许存在模糊
- 搜索关键词长度为3-5，允许1次模糊
- 搜索关键词长度大于5，允许最大2次模糊

高亮highlight

highlight 关键字: 可以让符合条件的文档中的关键词高亮。

highlight相关属性:

- pre_tags 前缀标签
- post_tags 后缀标签

- tags_schema 设置为styled可以使用内置高亮样式
- require_field_match 多字段高亮需要设置为false

示例数据

```
1 #指定ik分词器
2 PUT /products
3 {
4   "settings" : {
5     "index" : {
6       "analysis.analyzer.default.type": "ik_max_word"
7     }
8   }
9 }
10
11 PUT /products/_doc/1
12 {
13   "proId" : "2",
14   "name" : "牛仔男外套",
15   "desc" : "牛仔外套男装春季衣服男春装夹克修身体闲男生潮牌工装潮流头号青年春秋棒球服男 7705浅蓝常规 XL",
16   "timestamp" : 1576313264451,
17   "createTime" : "2019-12-13 12:56:56"
18 }
19
20 PUT /products/_doc/2
21 {
22   "proId" : "6",
23   "name" : "HLA海澜之家牛仔裤男",
24   "desc" : "HLA海澜之家牛仔裤男2019时尚有型舒适HKNAD3E109A 牛仔蓝(A9)175/82A(32)",
25   "timestamp" : 1576314265571,
26   "createTime" : "2019-12-18 15:56:56"
27 }
```

测试

```
1 GET /products/_search
2 {
3   "query": {
4     "term": {
5       "name": {
6         "value": "牛仔"
```

```
7   }
8   }
9   },
10  "highlight": {
11    "fields": {
12      "*": {}
13    }
14  }
15 }
```

自定义高亮html标签

可以在highlight中使用pre_tags和post_tags

```
1  GET /products/_search
2  {
3    "query": {
4      "term": {
5        "name": {
6          "value": "牛仔"
7        }
8      }
9    },
10   "highlight": {
11     "post_tags": ["/span>"],
12     "pre_tags": ["<span style='color:red'>"],
13     "fields": {
14       "*": {}
15     }
16   }
17 }
```

多字段高亮

```
1
2  GET /products/_search
3  {
4    "query": {
5      "term": {
6        "name": {
7          "value": "牛仔"
8        }
9      }
10   }
```

```
10 },
11 "highlight": {
12 "pre_tags": ["<font color='red'>"],
13 "post_tags": ["<font/>"],
14 "require_field_match": "false",
15 "fields": {
16 "name": {},
17 "desc": {}
18 }
19 }
20 }
21
```