主讲老师：Fox

> 有道笔记地址： https://note.youdao.com/s/3llHx6vE

1. 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 领域专用语言），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:

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

示例

```
#无条件查询，默认返回10条数据
GET /es_db/_search
{
    "query":{
        "match_all":{}
    }
}
```


```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 13,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "es_db",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : 1.0,
        "_source" : {
          "name" : "李四",
          "sex" : 1,
          "age" : 28,
          "address" : "广州荔湾大厦",
          "remark" : "java assistant"
        }
      },
      {
```

示例数据

```
#指定ik分词器
PUT /es_db
{
  "settings" : {
      "index" : {
          "analysis.analyzer.default.type": "ik_max_word"
      }
  }
}

# 创建文档,指定id
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": "王五",
"sex": 0,
"age": 26,
"address": "广州白云山公园",
"remark": "php developer"
}

PUT /es_db/_doc/4
{
"name": "赵六",
"sex": 0,
"age": 22,
"address": "长沙橘子洲",
"remark": "python assistant"
}

PUT /es_db/_doc/5
{
"name": "张龙",
"sex": 0,
"age": 19,
"address": "长沙麓谷企业广场",
"remark": "java architect assistant"
}
```

```
PUT /es_db/_doc/6
{
"name": "赵虎",
"sex": 1,
"age": 32,
"address": "长沙麓谷兴工国际产业园",
"remark": "java architect"
}

PUT /es_db/_doc/7
{
"name": "李虎",
"sex": 1,
"age": 32,
"address": "广州番禺节能科技园",
"remark": "java architect"
}

PUT /es_db/_doc/8
{
"name": "张星",
"sex": 1,
"age": 32,
"address": "武汉东湖高新区未来智汇城",
"remark": "golang developer"
}

```

1.1 match_all

使用match_all，匹配所有文档，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

```
GET /es_db/_search
等同于
```

```
GET /es_db/_search
{
"query":{
"match_all":{}
}
}
```

返回源数据_source

_source 关键字: 是一个数组,在数组中用来指定展示那些字段

```
# 返回指定字段
GET /es_db/_search
{
  "query": {
    "match_all": {}
  },
  "_source": ["name","address"]
}

#在查询中过滤
#不查看源数据，仅查看元字段
{
  "_source": false,
  "query": {
    ...
  }
}

#只看以obj.开头的字段
{
  "_source": "obj.*",
  "query": {
    ...
  }
}
```

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

```
GET /es_db/_search
{
  "query": {
    "match_all": {}
  },
  "size": 100
}

```

分页查询from&size

size：显示应该返回的结果数量，默认是 10

from：显示应该跳过的初始结果数量，默认是 0

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

```
GET /es_db/_search
{
  "query": {
    "match_all": {}
  },
  "from": 0,
  "size": 5
}
```

指定字段排序sort

注意：会让得分失效

```
GET /es_db/_search
```

```
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "age": "desc"
    }
  ]
}

#排序，分页
GET /es_db/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "age": "desc"
    }
  ],
  "from": 10,
  "size": 5
}
```

1.2 术语级别查询

术语级别查询（Term-Level Queries）指的是搜索内容不经过文本分析直接用于文本匹配，这个过程类似于数据库的SQL查询，搜索的对象大多是索引的非text类型字段。Elasticsearch 中的一些术语级别查询示例包括 term、terms 和 range 查询。

Term query术语查询

术语查询直接返回包含搜索内容的文档，常用来查询索引中某个类型为keyword的文本字段，类似于SQL的“=”查询，使用十分普遍。

注意：最好不要在term查询的字段中使用text字段，因为text字段会被分词，这样做既没有意义，还很有可能什么也查不到。

```
# 对bool，日期，数字，结构化的文本可以利用term做精确匹配
# term 精确匹配
GET /es_db/_search
{
  "query": {
    "term": {
      "age": {
        "value": 28
      }
    }
  }
}

# 思考： 查询广州白云是否有数据，为什么？
GET /es_db/_search
{
  "query":{
    "term": {
      "address": {
        "value": "广州白云"
      }
    }
  }
}

# 采用term精确查询，查询字段映射类型为keyword
GET /es_db/_search
{
  "query":{
    "term": {
      "address.keyword": {
        "value": "广州白云山公园"
      }
    }
  }
}


```

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。

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

```
GET /es_db/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "address.keyword": "广州白云山公园"
        }
      }
    }
  }
}
```

term处理多值字段时，term查询是包含，不是等于。

```
POST /employee/_bulk
{"index":{"_id":1}}
{"name":"小明","interest":["跑步","篮球"]}
{"index":{"_id":2}}
{"name":"小红","interest":["跳舞","画画"]}
{"index":{"_id":3}}
{"name":"小丽","interest":["跳舞","唱歌","跑步"]}

POST /employee/_search
{
  "query": {
    "term": {
      "interest.keyword": {
        "value": "跑步"
      }
    }
```

```
  }
}
```

Terms Query多术语查询

Terms query用于在指定字段上匹配多个词项（terms）。它会精确匹配指定字段中包含的任何一个词项。

```
POST /es_db/_search
{
  "query": {
    "terms": {
      "remark.keyword": ["java assistant", "java architect"]
    }
  }
}

```

exists query

在Elasticsearch中可以使用exists进行查询，以判断文档中是否存在对应的字段。

```
#查询索引库中存在remarks字段的文档数据
GET /es_db/_search
{
  "query": {
    "exists":
    {
      "field": "remark"
    }
  }
}
```

ids query

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

```
GET /es_db/_search
{
  "query": {
    "ids": {
      "values": [1,2]
    }
  }
}
```

range query范围查询

- 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/_search
{
  "query": {
    "range": {
      "age": {
        "gte": 25,
        "lte": 28
      }
    }
  }
}

#日期范围比较
DELETE /product
POST /product/_bulk
{"index":{"_id":1}}
{"price":100,"date":"2021-01-01","productId":"XHDK-1293"}
{"index":{"_id":2}}
```

```
{"price":200,"date":"2022-01-01","productId":"KDKE-5421"}

GET /product/_mapping

GET /product/_search
{
  "query": {
    "range": {
      "date": {
        "gte": "now-2y"
      }
    }
  }
}
```

prefix query前缀查询

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是否以所指定的前缀开头。

```
GET /es_db/_search
{
  "query": {
    "prefix": {
      "address": {
        "value": "广州"
      }
    }
  }
}
```

wildcard query通配符查询

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。

```
GET /es_db/_search
{
  "query": {
    "wildcard": {
      "address": {
        "value": "*白*"
      }
    }
  }
}
```

fuzzy query模糊查询

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；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。
  - 该参数默认值为0，即不开启模糊查询; fuzzy 模糊查询 最大模糊错误必须在0-2之间
- 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。

```
GET /es_db/_search
{
  "query": {
    "fuzzy": {
      "address": {
        "value": "白运山",
        "fuzziness": 1
      }
    }
  }
}

```

1.3 全文检索

全文检索查询（Full Text Queries）和术语级别查询（Term-Level Queries）是 Elasticsearch 中搜索和检索数据的两种不同方法。

全文检索查询旨在基于相关性搜索和匹配文本数据。这些查询会对输入的文本进行分析，将其拆分为词项（单个单词），并执行诸如分词、词干处理和标准化等操作。Elasticsearch 中的一些全文检索查询示例包括 match、match_phrase 和 multi_match 查询。

全文检索的关键特点：

- 对输入的文本进行分析，并根据分析后的词项进行搜索和匹配。全文检索查询会对输入的文本进行分析，将其拆分为词项，并基于这些词项进行搜索和匹配操作。
- 以相关性为基础进行搜索和匹配。全文检索查询使用相关性算法来确定文档与查询的匹配程度，并按照相关性进行排序。相关性可以基于词项的频率、权重和其他因素来计算。
- 全文检索查询适用于包含自由文本数据的字段，例如文档的内容、文章的正文或产品描述等。

match query匹配查询

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)

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

```
#match 分词后or的效果
GET /es_db/_search
{
  "query": {
    "match": {
      "address": "广州白云山公园"
    }
  }
}

# 分词后 and的效果
GET /es_db/_search
{
  "query": {
    "match": {
      "address": {
        "query": "广州白云山公园",
        "operator": "and"
      }
    }
  }
}

```

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。

```
# 最少匹配广州，公园两个词
GET /es_db/_search
{
  "query": {
    "match": {
      "address": {
        "query": "广州公园",
```

```
        "minimum_should_match": 2
      }
    }
  }
}
```

对于match查询，其底层逻辑的概述：

1. 分词：首先，输入的查询文本会被分词器进行分词。分词器会将文本拆分成一个个词项（terms），如单词、短语或特定字符。分词器通常根据特定的语言规则和配置进行操作。
2. 倒排索引：ES使用倒排索引来加速搜索过程。倒排索引是一种数据结构，它将词项映射到包含这些词项的文档。每个词项都有一个对应的倒排列表，其中包含了包含该词项的所有文档的引用。
3. 匹配计算：一旦查询被分词，ES将根据查询的类型和参数计算文档与查询的匹配度。对于match查询，ES将比较查询的词项与倒排索引中的词项，并计算文档的相关性得分。相关性得分衡量了文档与查询的匹配程度。
4. 结果返回：根据相关性得分，ES将返回最匹配的文档作为搜索结果。搜索结果通常按照相关性得分进行排序，以便最相关的文档排在前面。

multi_match query 多字段查询

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

```
GET /es_db/_search
{
  "query": {
    "multi_match": {
      "query": "长沙张龙",
      "fields": [
        "address",
        "name"
      ]
    }
  }
}
```

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。

match_phrase query短语查询

短语搜索(match phrase)会对搜索文本进行文本分析，然后到索引中寻找搜索的每个分词并要求分词相邻，你可以通过调整slop参数设置分词出现的最大间隔距离。match_phrase 会将检索关键词分词。

```
GET /es_db/_search
{
  "query": {
    "match_phrase": {
      "address": "广州白云山"
    }
  }
}
GET /es_db/_search
{
  "query": {
    "match_phrase": {
      "address": "广州白云"
    }
  }
}
```

思考：为什么查询广州白云山有数据，广州白云没有数据?

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。

```
POST _analyze
{
    "analyzer":"ik_max_word",
    "text":"广州白云山"
}
#结果
{
  "tokens" : [
    {
      "token" : "广州",
      "start_offset" : 0,
      "end_offset" : 2,
      "type" : "CN_WORD",
      "position" : 0
    },
    {
      "token" : "白云山",
      "start_offset" : 2,
      "end_offset" : 5,
      "type" : "CN_WORD",
      "position" : 1
    },
    {
      "token" : "白云",
      "start_offset" : 2,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 2
    },
    {
      "token" : "云山",
      "start_offset" : 3,
      "end_offset" : 5,
      "type" : "CN_WORD",
      "position" : 3
    }
```

```
  ]
}
```

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。

```
#广州云山分词后相隔为2，可以匹配到结果
GET /es_db/_search
{
  "query": {
    "match_phrase": {
      "address": {
        "query": "广州云山",
        "slop": 2
      }
    }
  }
}

```

query_string query

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**

```
# AND 要求大写
GET /es_db/_search
{
  "query": {
    "query_string": {
      "query": "赵六 AND 橘子洲"
    }
```

```
  }
}
```

- **指定单个字段查询**

```
#Query String
GET /es_db/_search
{
  "query": {
    "query_string": {
      "default_field": "address",
      "query": "白云山 OR 橘子洲"
    }
  }
}
```

- **指定多个字段查询**

```
GET /es_db/_search
{
  "query": {
    "query_string": {
      "fields": ["name","address"],
      "query": "张三 OR (广州 AND 王五)"
    }
  }
}

```

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

```
#simple_query_string 默认的operator是OR
GET /es_db/_search
{
  "query": {
    "simple_query_string": {
      "fields": ["name","address"],
      "query": "广州公园",
      "default_operator": "AND"
    }
  }
}

GET /es_db/_search
{
  "query": {
    "simple_query_string": {
      "fields": ["name","address"],
      "query": "广州 + 公园"
    }
  }
}
```

1.4 bool query布尔查询

布尔查询可以按照布尔逻辑条件组织多条查询语句，只有符合整个布尔条件的文档才会被搜索出来。在布尔条件中，可以包含两种不同的上下文。

1. **搜索上下文(query context)**：使用搜索上下文时，Elasticsearch需要计算每个文档与搜索条件的相关度得分，这个得分的计算需使用一套复杂的计算公式，有一定的性能开销，带文本分析的全文检索的查询语句很适合放在搜索上下文中。
2. **过滤上下文(filter context)**：使用过滤上下文时，Elasticsearch只需要判断搜索条件跟文档数据是否匹配，例如使用Term query判断一个值是否跟搜索内容一致，使用Range query判断某数据是否位于某个区间等。过滤上

下文的查询不需要进行相关度得分计算，还可以使用缓存加快响应速度，很多术语级查询语句都适合放在过滤上下文中。

布尔查询一共支持4种组合类型:

类型	说明
must	可包含多个查询条件，每个条件均满足的文档才能被搜索到，每次查询需要计算相关度得分，属于搜索上下文
should	可包含多个查询条件，不存在must和fiter条件时，至少要满足多个查询条件中的一个，文档才能被搜索到，否则需满足的条件数量不受限制,匹配到的查询越多相关度越高，也属于搜索上下文
filter	可包含多个过滤条件，每个条件均满足的文档才能被搜索到，每个过滤条件不计算相关度得分，结果在一定条件下会被缓存， 属于过滤上下文
must_not	可包含多个过滤条件，每个条件均不满足的文档才能被搜索到，每个过滤条件不计算相关度得分，结果在一定条件下会被缓存， 属于过滤上下文

示例

```
PUT /books
{
  "settings": {
    "number_of_replicas": 1,
    "number_of_shards": 1
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "long"
      },
      "title": {
        "type": "text",
        "analyzer": "ik_max_word"
      },
      "language": {
        "type": "keyword"
```

```
      },
      "author": {
        "type": "keyword"
      },
      "price": {
        "type": "double"
      },
      "publish_time": {
        "type": "date",
        "format": "yyy-MM-dd"
      },
      "description": {
        "type": "text",
        "analyzer": "ik_max_word"
      }
    }
  }
}

POST /_bulk
{"index":{"_index":"books","_id":"1"}}
{"id":"1", "title":"Java编程思想", "language":"java", "author":"Bruce Eckel", "price":70.20, "publish_time":"2007-10-01", "description":"Java学习必读经典，殿堂级著作！赢得了全球程序员的广泛赞誉。"}
{"index":{"_index":"books","_id":"2"}}
{"id":"2","title":"Java程序性能优化","language":"java","author":"葛一鸣","price":46.5,"publish_time":"2012-08-01","description":"让你的Java程序更快、更稳定。深入剖析软件设计层面、代码层面、JVM虚拟机层面的优化方法"}
{"index":{"_index":"books","_id":"3"}}
{"id":"3","title":"Python科学计算","language":"python","author":"张若愚","price":81.4,"publish_time":"2016-05-01","description":"零基础学python，光盘中作者独家整合开发winPython运行环境，涵盖了Python各个扩展库"}
{"index":{"_index":"books","_id":"4"}}
{"id":"4", "title":"Python基础教程", "language":"python", "author":"Helant", "price":54.50, "publish_time":"2014-03-01", "description":"经典的Python入门教程，层次鲜明，结构严谨，内容翔实"}
{"index":{"_index":"books","_id":"5"}}
{"id":"5","title":"JavaScript高级程序设计","language":"javascript","author":"Nicholas C. Zakas","price":66.4,"publish_time":"2012-10-01","description":"JavaScript技术经典名著"}


GET /books/_search
```

```
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": "java编程"
          }
        },{
          "match": {
            "description": "性能优化"
          }
        }
      ]
    }
  }
}

GET /books/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "title": "java编程"
          }
        },{
          "match": {
            "description": "性能优化"
          }
        }
      ],
      "minimum_should_match": 1
    }
  }
}
GET /books/_search
{
  "query": {
```

```
    "bool": {
      "filter": [
        {
          "term": {
            "language": "java"
          }
        },
        {
          "range": {
            "publish_time": {
              "gte": "2010-08-01"
            }
          }
        }
      ]
    }
  }
}
```

1.5 highlight高亮

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

```
#指定ik分词器
PUT /products
{
  "settings" : {
      "index" : {
```

```
            "analysis.analyzer.default.type": "ik_max_word"
        }
    }
}

PUT /products/_doc/1
{
  "proId" : "2",
  "name" : "牛仔男外套",
  "desc" : "牛仔外套男装春季衣服男春装夹克修身休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",
  "timestamp" : 1576313264451,
  "createTime" : "2019-12-13 12:56:56"
}

PUT /products/_doc/2
{
  "proId" : "6",
  "name" : "HLA海澜之家牛仔裤男",
  "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)",
  "timestamp" : 1576314265571,
  "createTime" : "2019-12-18 15:56:56"
}
```

测试

```
GET /products/_search
{
  "query": {
    "term": {
      "name": {
        "value": "牛仔"
      }
    }
  },
  "highlight": {
    "fields": {
      "*":{}
    }
```

```
  }
}
```

自定义高亮html标签

可以在highlight中使用pre_tags和post_tags

```
GET /products/_search
{
  "query": {
    "term": {
      "name": {
        "value": "牛仔"
      }
    }
  },
  "highlight": {
    "post_tags": ["</span>"],
    "pre_tags": ["<span style='color:red'>"],
    "fields": {
      "*":{}
    }
  }
}
```

多字段高亮

```

GET /products/_search
{
  "query": {
    "term": {
      "name": {
        "value": "牛仔"
      }
```

```
    }
  },
  "highlight": {
    "pre_tags": ["<font color='red'>"],
    "post_tags": ["<font/>"],
    "require_field_match": "false",
    "fields": {
      "name": {},
      "desc": {}
    }
  }
}

```

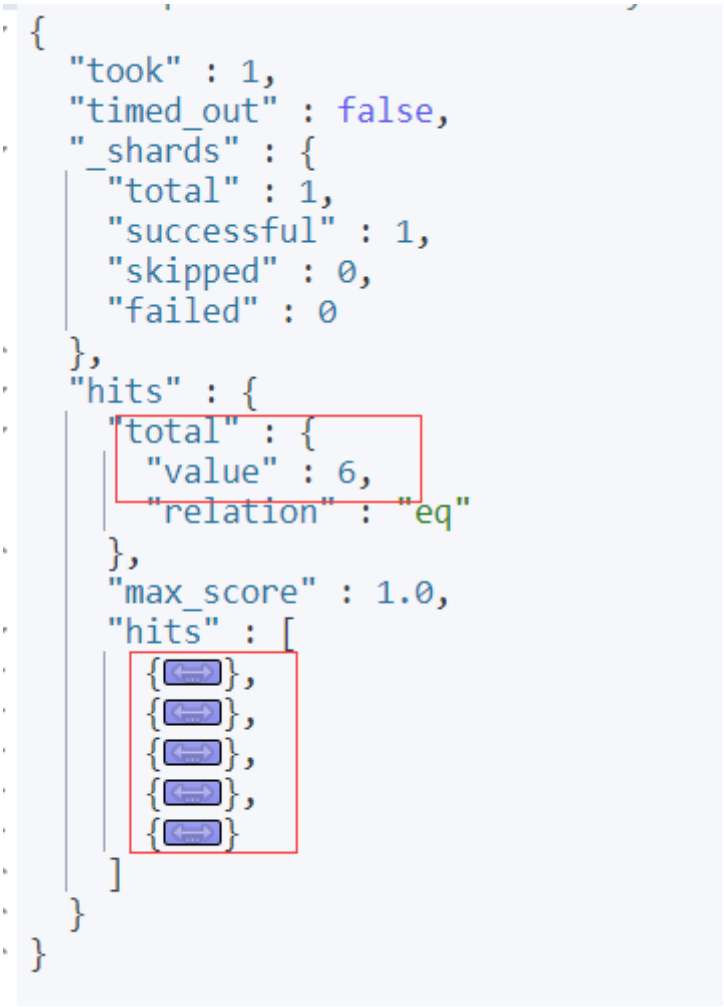
2. ES 深度分页问题及针对不同需求下的解决方案

2.1 什么是深度分页

分页问题是Elasticsearch中最常见的查询场景之一，正常情况下分页代码如实下面这样的：

```
# 查询第一页5条数据
GET /es_db/_search
{
  "query": {
    "match_all": {}
  },
  "from": 0,
  "size": 5
}
```

输出结果如下图：



但是如果我们查询的数据页数特别大，当from + size大于10000的时候，就会出现问题，如下图报错信息所示：

ES通过参数index.max_result_window用来限制单次查询满足查询条件的结果窗口的大小，其默认值为10000。

2.2 深度分页会带来什么问题

ES分页查询流程大致如下：

1. 数据存储在各个分片中，协调节点将查询请求转发给各个节点，当各个节点执行搜索后，将排序后的前N条数据返回给协调节点。
2. 协调节点汇总各个分片返回的数据，再次排序，最终返回前N条数据给客户端。
3. 这个流程会导致一个深度分页的问题，也就是翻页越多，性能越差，甚至导致ES出现OOM。

在分布式系统中，对结果排序的成本随分页的深度成指数上升。

从10万名高考生中查询成绩为的10001-10100位的100名考生的信息。

从上面案例中不难看出，每次有序的查询都会在每个分片中执行单独的查询，然后进行数据的二次排序，而这个二次排序的过程是发生在heap中的，也就是说当你单次查询的数量越大，那么堆内存中汇总的数据也就越多，对内存的压力也就越大。这里的单次查询的数据量取决于你查询的是第几条数据而不是查询了几条数据，比如你希望查询的是第10001-10100这一百条数据，但是ES必须将前10100

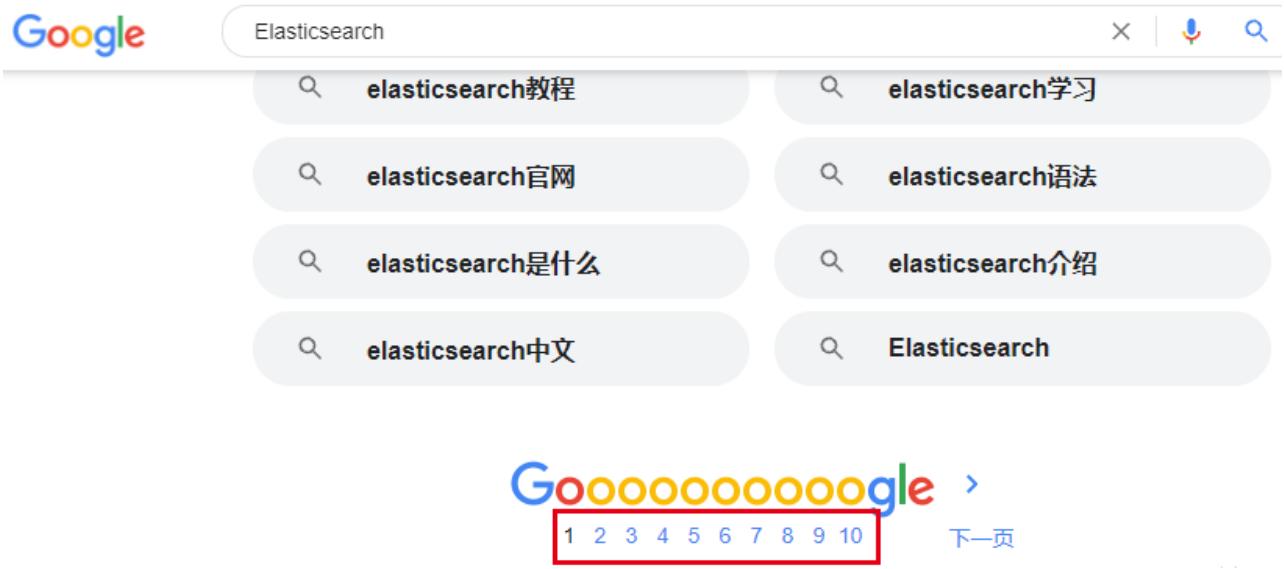
全部取出进行二次查询。因此，如果查询的数据排序越靠后，就越容易导致OOM（Out Of Memory）情况的发生，频繁的深分页查询会导致频繁的FGC。

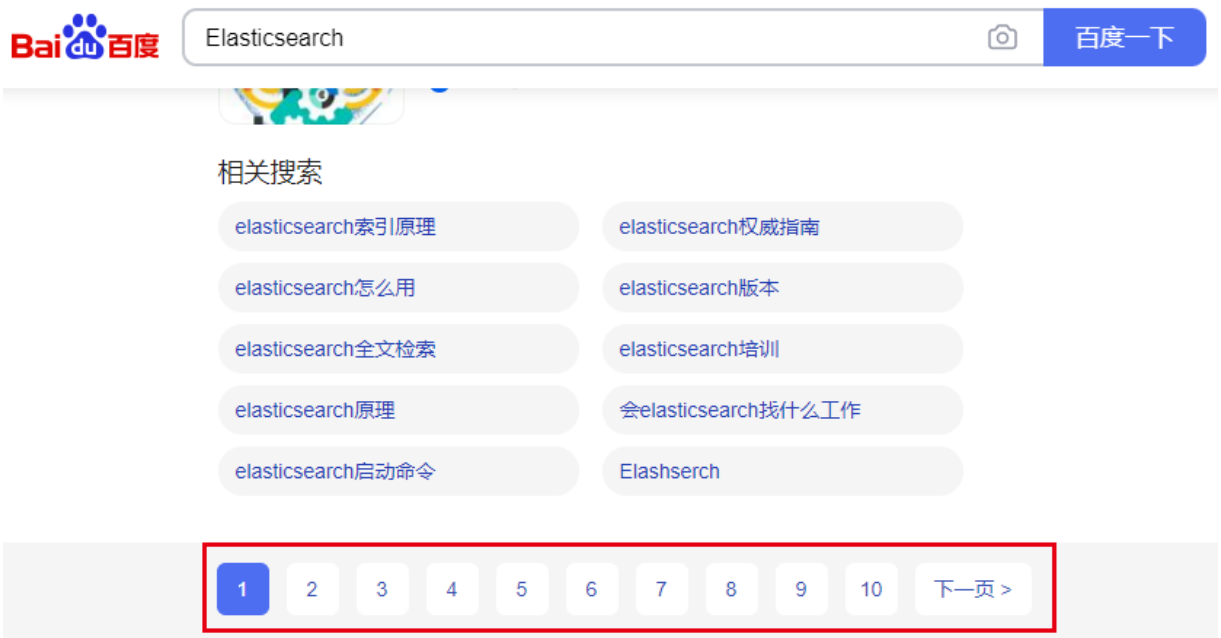
ES为了避免用户在不了解其内部原理的情况下而做出错误的操作，设置了一个阈值，即max_result_window，其默认值为10000，其作用是为了保护堆内存不被错误操作导致溢出。

2.3 深度分页问题的常见解决方案

尝试避免使用深度分页

解决深度分页问题最好的办法就是避免使用深度分页。谷歌、百度目前作为全球和国内做大的搜索引擎不约而同的在分页条中删除了“跳页”功能，其目的就是为了避免用户使用深度分页检索。





淘宝虽然没有删除“跳页”功能，但不管我们搜索什么内容，只要商品结果足够多，返回的商品列表都是仅展示前100页的数据，其本质和ES中的max_result_window作用是一样的，都是限制你去搜索更深页数的数据。

手机端APP就更不用说了，直接是下拉加载更多，连分页条都没有，相当于你只能点击“下一页”。

滚动查询：Scroll Search

scroll滚动搜索是先搜索一批数据，然后下次再搜索下一批数据，以此类推，直到搜索出全部的数据来。

scroll搜索会在第一次搜索的时候，保存一个当时的视图快照，之后只会基于该视图快照搜索数据，如果在搜索期间数据发生了变更，用户是看不到变更的数据的。因此，滚动查询不适合实时性要求高的搜索场景。

官方已不推荐使用滚动查询进行深度分页查询，因为无法保存索引状态。

适合场景

单个滚动搜索请求中检索大量结果，即非“C端业务”场景

使用

1）第一次进行scroll查询：

```
#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1分钟，也就是本次快照的结果缓存起来的有效时间是1分钟。
#这里由于测试数据量不够，所以size值设置为2。
#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。
GET /es_db/_search?scroll=1m
{
    "query": { "match_all": {}},
    "size":  2
}
```

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。

2）从第二次查询开始，每次查询都要指定_scroll_id参数：

```
# scroll_id 的值就是上一个请求中返回的 _scroll_id 的值
GET /_search/scroll
{
    "scroll": "1m",
    "scroll_id" :
"FGluY2x1ZGVfY29udGV4dF91dWlkDXF1ZXJ5QW5kRmV0Y2gBFmNwcVdjblRxUzVhZXlicG9HeU02bWcAAAAAAA
BmzRY2YlV3Z0o5VVNTdWJobkE5Z3MtXzJB"
}
```

查询结果：

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。

删除游标scroll

scroll超过超时后，搜索上下文会自动删除。然而，保持scroll打开是有代价的，因此一旦不再使用，就应明确清除scroll上下文

```
DELETE /_search/scroll
{
    "scroll_id" :
"FGluY2x1ZGVfY29udGV4dF91dWlkDXF1ZXJ5QW5kRmV0Y2gBFmNwcVdjblRxUzVhZXlicG9HeU02bWcAAAAAAA
BmzRY2YlV3Z0o5VVNTdWJobkE5Z3MtXzJB"
}
```

注意事项

- scroll滚动查询不适合实时性要求高的查询场景，比较适合数据迁移的场景。
- scroll查询完毕后，要手动清理掉 scroll_id 。虽然ES有自动清理机制，但是 srcoll_id 的存在会耗费大量的资源来保存一份当前查询结果集映像，并且会占用文件描述符。

官方建议：ES7之后，不再建议使用scroll API进行深度分页。如果要分页检索超过 Top 10,000+ 结果时，推荐使用：PIT + search_after。

search_after

参考文档：https://www.elastic.co/guide/en/elasticsearch/reference/7.17/paginate-search-results.html#search-after

scroll API适用于高效的深度滚动，但滚动上下文成本高昂，不建议将其用于实时用户请求。而search_after参数通过提供一个活动光标来规避这个问题。这样可以使用上一页的结果来帮助检索下一页。

search_after 分页查询可以简单概括为如下几个步骤：

1）获取索引的pit

使用 search_after 需要具有相同查询和排序值的多个搜索请求。 如果在这些请求之间发生刷新，结果的顺序可能会发生变化，从而导致跨页面的结果不一致。 为防止出现这种情况，可以创建一个时间点 (PIT) 以保留搜索中的当前索引状态。Point In Time（PIT）是 Elasticsearch 7.10 版本之后才有的新特性。

```
# 创建一个时间点(PIT)来保存搜索期间的当前索引状态
POST /es_db/_pit?keep_alive=1m
#返回结果，会返回一个PID的值
{
  "id" : "39K1AwEFZXNfZGIWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAWdkhjbE9YNVRTMUNDcWNQQVR2ZXYzdwAAAAAAAA9jhZvaGpLSDlzVVMxbW5idG5DZ0xEUHFRAAEWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAA"
}

```

2） 根据pit首次查询

根据pit查询的时候，不用指定索引的名词

```
GET /_search
{
  "query": {
        "match_all": {}
    },
  "pit": {
            "id": "39K1AwEFZXNfZGIWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAWdkhjbE9YNVRTMUNDcWNQQVR2ZXYzdwAAAAAAAA9jhZvaGpLSDlzVVMxbW5idG5DZ0xEUHFRAAEWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAA",
            "keep_alive": "1m"
  },
  "size": 2,
  "sort": [
        {"_id": "asc"}
    ]
}

```

返回结果：


```
{
  "pit_id" : "39K1AwEFZXNfZGIWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAWdkhjbE9YNVRTMUNDcWNQQVR2ZXYzdwAAAAAAAAA7hRZvaGpLSDlzVVMxbW5idG5DZ0xEUHFRAAEWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAA",
  "took" : 16,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 5,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [
      {
        "_index" : "es_db",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : null,
        "_source" : {
          "name" : "李四",
          "sex" : 1,
          "age" : 28,
          "address" : "广州荔湾大厦",
          "remark" : "java assistant"
        },
        "sort" : [
          "2",
          0
        ]
      },
      {
```

```
        "_index" : "es_db",
        "_type" : "_doc",
        "_id" : "3",
        "_score" : null,
        "_source" : {
          "name" : "王五",
          "sex" : 0,
          "age" : 26,
          "address" : "广州白云山公园",
          "remark" : "php developer"
        },
        "sort" : [
          "3",
          1
        ]
      }
    ]
  }
}
```

3）根据search_after和pit进行翻页查询

要获得下一页结果，请使用最后一次命中的排序值（包括 tiebreaker）作为 search_after 参数重新运行先前的搜索。 如果使用 PIT，请在 pit.id 参数中使用最新的 PIT ID。 搜索的查询和排序参数必须保持不变。

```
#search_after指定为上一次查询返回的sort值。
GET /_search
{
  "query": {
        "match_all": {}
    },
  "pit": {
            "id":
"39K1AwEFZXNfZGIWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAWdkhjbE9YNVRTMUNDcWNQQVR2ZXYzdwAAAAAAAA
A9jhZvaGpLSDlzVVMxbW5idG5DZ0xEUHFRAAEWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAA",
            "keep_alive": "1m"
  },
  "size": 2,
  "sort": [
```

```
        {"_id": "asc"}
    ],
  "search_after": [
    5
  ]
}
```

返回结果：

```
{
  "pit_id" : "39K1AwEFZXNfZGIWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAWdkhjbE9YNVRTMUNDcWNQQVR2ZXYzdwAAAAAAAAA8wxZvaGpLSDlzVVMxbW5idG5DZ0xEUHFRAAEWZTN2N2Nrdk5RRjY3QjBma1h5aFRodwAA",
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 5,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [
      {
        "_index" : "es_db",
        "_type" : "_doc",
        "_id" : "4",
        "_score" : null,
        "_source" : {
          "name" : "赵六",
          "sex" : 0,
          "age" : 22,
          "address" : "长沙橘子洲",
          "remark" : "python assistant"
```

```
        },
        "sort" : [
          "4"
        ]
      },
      {
        "_index" : "es_db",
        "_type" : "_doc",
        "_id" : "5",
        "_score" : null,
        "_source" : {
          "name" : "张龙",
          "sex" : 0,
          "age" : 19,
          "address" : "长沙麓谷企业广场",
          "remark" : "java architect assistant"
        },
        "sort" : [
          "5"
        ]
      }
    ]
  }
}
```

2.4 总结

分页方式	性能	优点	缺点	适用场景
from + size	低	灵活性好，实现简单，支持随机翻页	受制于max_result_window设置，不能无限制翻页； 存在深度翻译问题，越往后翻译越慢。	数据量比较小，能容忍深度分页问题
scroll	中	解决了深度分页问题	scroll查询的相应数据是非实时的，如果遍历过程中插	海量数据的导出，需要查询海量结果集的数据

			入新的数据，是查询不到的； 保留上下文需要足够的堆内存空间。	
search_after	高	性能最好，不存在深度分页问题，能够反映数据的实时变更	实现复杂，需要有一个全局唯一的字段连续分页的实现会比较复杂，因为每一次查询都需要上次查询的结果，它不适用于大幅度跳页查询	海量数据的分页