> 主讲老师：Fox
> 有道云笔记地址：https://note.youdao.com/s/8WkK3m2H

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。

> 官方文档： https://www.elastic.co/guide/en/elasticsearch/reference/8.14/query-dsl.html

基本语法:

```
GET /<index_name>/_search {json请求体数据}
```

示例数据准备

```
DELETE /employee
PUT /employee
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "keyword"
      },
      "sex": {
        "type": "integer"
      },
      "age": {
        "type": "integer"
      },
      "address": {
        "type": "text",
        "analyzer": "ik_max_word",
        "fields": {
          "keyword": {
            "type": "keyword"
          }
        }
      },
      "remark": {
        "type": "text",
        "analyzer": "ik_smart",
        "fields": {
          "keyword": {
            "type": "keyword"
          }
        }
      }
    }
  }
}
```

```
POST /employee/_bulk
{"index":{"_index":"employee","_id":"1"}}
{"name":"张三","sex":1,"age":25,"address":"广州天河公园","remark":"java developer"}
{"index":{"_index":"employee","_id":"2"}}
{"name":"李四","sex":1,"age":28,"address":"广州荔湾大厦","remark":"java assistant"}
{"index":{"_index":"employee","_id":"3"}}
{"name":"王五","sex":0,"age":26,"address":"广州白云山公园","remark":"php developer"}
{"index":{"_index":"employee","_id":"4"}}
{"name":"赵六","sex":0,"age":22,"address":"长沙橘子洲","remark":"python assistant"}
{"index":{"_index":"employee","_id":"5"}}
{"name":"张龙","sex":0,"age":19,"address":"长沙麓谷企业广场","remark":"java architect assistant"}
{"index":{"_index":"employee","_id":"6"}}
{"name":"赵虎","sex":1,"age":32,"address":"长沙麓谷兴工国际产业园","remark":"java architect"}
```

1.1 match_all ——匹配所有文档

match_all查询是一个特殊的查询类型，它用于匹配索引中的所有文档，而不考虑任何特定的查询条件。

基本语法

```
GET /<your-index-name>/_search
{
 "query": {
    "match_all": {}
 }
}
```

高级用法

例如，如果您想要返回索引中的前10个文档，并且按照文档的评分进行排序，您可以使用以下查询：

```
GET /<your-index-name>/_search
{
 "query": {
    "match_all": {}
 },
 "size": 10,
 "sort": [
    {"_score": {"order": "desc"}}
 ]
}

```

_source的用法

```
#不查看源数据，仅查看元字段

GET /<your-index-name>/_search
{
  "query": {
    "match_all": {}
  },
  "_source": false
}

# 返回指定字段
GET /<your-index-name>/_search
{
  "query": {
    "match_all": {}
  },
  "_source": ["field1","field2"]
}

#只看以obj.开头的字段
GET /<your-index-name>/_search
{
  "query": {
    "match_all": {}
  },
  "_source": "obj.*"
}


```

示例

- size返回指定条数

```
GET /employee/_search
{
  "query": {
    "match_all": {}
  },
  "size": 3
}


```

- from&size分页查询

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


```

- sort指定字段排序

```
# 根据age排序
GET /employee/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "age": "desc"
    }
  ]
}

# 排序的同时进行分页
GET /employee/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "age": "desc"
    }
  ],
  "from": 2,
  "size": 5
}


```

- _source返回源数据

```
GET /employee/_search
{
  "query": {
    "match_all": {}
  },
  "_source": ["name","address"]
}

```

1.2 精确匹配

精确匹配是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。**此类检索主要应用于结构化数据，如ID、状态和标签等。**

term——单字段精确匹配查询

term检索主要应用于单字段精准匹配的场景。在实战过程中，需要避免将term检索应用于text类型的检索。进一步说，term检索针对的是非text类型，用于text类型时并不会报错，但检索结果一般会达不到预期。

基本语法

在Elasticsearch 8.x中，term查询用于执行精确匹配查询，它适用于未经过分词处理的keyword字段类型。term查询的基本语法如下：

```
GET /{index_name}/_search
{
 "query": {
    "term": {
      "{field.keyword}": {
        "value": "your_exact_value"
      }
    }
 }
}
```

这里的{index_name}是你要查询的索引名称，{field.keyword}是你要匹配的字段名称，.keyword后缀表示该字段是一个keyword类型，用于存储精确匹配的数据。"value"是你要精确匹配的值。

示例

对bool，日期，数字，结构化的文本可以利用term做精确匹配

```
# 查询姓名为张三的员工信息
GET /employee/_search
{
  "query": {
    "term": {
      "name": {
        "value": "张三"
      }
    }
  }
}
```

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。

```
# 思考： 查询广州白云是否有数据，为什么？
GET /employee/_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 /employee/_search
{
  "query":{
    "term": {
      "address.keyword": {
        "value": "广州白云山公园"
      }
    }
  }
}


```

term处理多值字段(数组)时，term查询是包含，不是等于。

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

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

terms——多值精确匹配

terms检索主要应用于多值精准匹配场景，它允许用户在单个查询中指定多个词条来进行精确匹配。这种查询方式适合从文档中查找包含多个特定值的字段，例如筛选出具有多个特定标签或状态的项目。而terms检索是针对未分析的字段进行精确匹配的，因此它在处理关键词、数字、日期等结构化数据时表现良好。

基本语法

在Elasticsearch 8.x中，进行多字段精确匹配时，可以使用terms查询。terms查询允许你指定一个字段，并匹配该字段中的多个精确值。

基本语法如下：

```
GET /<index_name>/_search
{
 "query": {
    "terms": {
      "<field_name>": [
        "value1",
        "value2",
        "value3",
        ...
      ]
    }
 }
}
```

> <index_name> 是你想要查询的索引名称。
> <field_name> 是你想要对其执行terms查询的字段名。
> 方括号内的值列表是你希望在查询中匹配的字段值。

示例

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

range——范围查询

range检索是Elasticsearch中一种针对指定字段值在给定范围内的文档的检索类型。这种查询适合对数字、日期或其他可排序数据类型的字段进行范围筛选。range检索支持多种比较操作符，如大于(gt)、大于等于(gte)、小于(lt)和小于等于(lte)等，可以实现灵活的区间查询。

基本语法

在Elasticsearch 8.x版本中，range查询的基本语法如下：

```
GET /<index_name>/_search
{
 "query": {
    "range": {
      "<field_name>": {
        "gte": <lower_bound>,
        "lte": <upper_bound>,
        "gt": <greater_than_bound>,
        "lt": <less_than_bound>
      }
    }
 }
}


```

> <index_name> 是你想要查询的索引名称。
> <field_name> 是你想要对其执行range查询的字段名。
> gte 表示大于或等于（Greater Than or Equal）。
> lte 表示小于或等于（Less Than or Equal）。
> gt 表示严格大于（Greater Than）。
> lt 表示严格小于（Less Than）。
> <lower_bound>, <upper_bound>, <greater_than_bound>, <less_than_bound> 是指定的数值边界。

示例

- 查询年龄在25到28的员工

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

```

- **日期范围查询**

1) 生成测试数据

假设我们正在创建一个笔记应用，每条笔记都有一个创建日期。

```
PUT /notes
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  },
  "mappings": {
    "properties": {
      "title": {"type": "text"},
      "content": {"type": "text"},
      "created_at": {"type": "date", "format": "yyyy-MM-dd HH:mm:ss"}
    }
  }
}

POST /notes/_bulk
{"index":{"_id":"1"}}
{"title":"Note 1","content":"This is the first note.","created_at":"2023-07-01 12:00:00"}
{"index":{"_id":"2"}}
{"title":"Note 2","content":"This is the second note.","created_at":"2023-07-05 15:30:00"}
{"index":{"_id":"3"}}
{"title":"Note 3","content":"This is the third note.","created_at":"2023-07-10 08:45:00"}
{"index":{"_id":"4"}}
{"title":"Note 4","content":"This is the fourth note.","created_at":"2023-07-15 20:15:00"}


```

2）使用range查询来查找在特定日期范围内的笔记。

假设我们想找出在2023年7月5日和2023年7月10日之间的所有笔记。

```
POST /notes/_search
{
  "query": {
    "range": {
      "created_at": {
        "gte": "2023-07-05 00:00:00",
        "lte": "2023-07-10 23:59:59"
      }
    }
  }
}
```

Elasticsearch支持日期数学表达式，允许在查询和聚合中使用相对时间点。以下是一些常见的日期数学表达式的示例和解释：

- now：当前时间点。
- now-1d：从当前时间点向前推1天的时间点。
- now-1w：从当前时间点向前推1周的时间点。
- now-1M：从当前时间点向前推1个月的时间点。
- now-1y：从当前时间点向前推1年的时间点。
- now+1h：从当前时间点向后推1小时的时间点。

```
POST /product/_bulk
{"index":{"_id":1}}
{"price":100,"date":"2023-01-01","productId":"XHDK-1293"}
{"index":{"_id":2}}
{"price":200,"date":"2022-01-01","productId":"KDKE-5421"}


# 返回所有在当前时间点前两年内的产品文档。
GET /product/_search
{
  "query": {
    "range": {
      "date": {
        "gte": "now-2y"
      }
    }
  }
}

```

exists——是否存在查询

exists检索在Elasticsearch中用于筛选具有特定字段值的文档。这种查询类型适用于检查文档中是否存在某个字段，或者该字段是否包含非空值。通过使用exists检索，你可以有效地过滤掉缺少关键信息的文档，从而专注于包含所需数据的结果。应用场景包括但不限于数据完整性检查、查询特定属性的文档以及对可选字段进行筛选等。

基本语法

```
GET /<index_name>/_search
{
 "query": {
    "exists": {
      "field": "missing_field"
    }
 }
}
```

示例

查询索引库中存在remark字段的文档

```
GET /employee/_search
{
  "query": {
    "exists":
    {
      "field": "remark"
    }
  }
}
```

ids——根据一组id查询

IDs检索也是一种常用的Elasticsearch查询方法，它允许我们基于给定的ID组快速召回相关数据，从而实现高效的文档检索。

基本语法

在Elasticsearch 8.x中，ids查询用于返回具有指定ID列表的文档。这个查询是检索特定文档的有效方式，特别是当你已经知道具体的文档ID时。

基本语法如下：

```
GET /<index_name>/_search
{
 "query": {
    "ids": {
      "values": ["id1", "id2", "id3", ...]
    }
 }
}
```

示例

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

prefix——前缀匹配

prefix会对分词后的term进行前缀搜索。

- 它不会对要搜索的字符串分词，传入的前缀就是想要查找的前缀
- 默认状态下，前缀查询不做相关性分数计算，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。

prefix的原理：

需要遍历所有倒排索引，并比较每个词项是否以所搜索的前缀开头。

基本语法

在Elasticsearch 8.x中，prefix查询用于搜索那些在指定字段中以特定前缀开始的文档。这种查询通常用于自动补全或搜索功能，其中用户输入的搜索词可能是更长文本的一部分。

基本语法如下：

```
GET /<index_name>/_search
{
 "query": {
    "prefix": {
      "your_field_name": {
        "value": "your_prefix_string"
      }
    }
 }
}
```

需要注意的是，这种查询方式仅适用于关键字类型(keyword)的字段。

示例

```
# 思考：能否查到数据？
GET /employee/_search
{
  "query": {
    "prefix": {
      "address": {
        "value": "广州白云山"
      }
    }
  }
}

GET /employee/_search
{
  "query": {
    "prefix": {
      "address.keyword": {
        "value": "广州白云山"
      }
    }
  }
}


```

wildcard——通配符匹配

wildcard检索是Elasticsearch中一种支持通配符匹配的查询类型，它允许在检索时使用通配符表达式来匹配文档的字段值。通配符包括两种。

- 星号(*)：表示零或多个字符，可用于匹配任意长度的字符串。
- 问号(?)：表示一个字符，用于匹配任意单个字符。

wildcard检索适用于对部分已知内容的文本字段进行模糊检索。例如，在文件名或产品型号等具有一定规律的字段中，使用通配符检索可以方便地找到满足特定模式的文档。

请注意，通配符查询可能会导致较高的计算负担，因此在实际应用中应谨慎使用，尤其是在涉及大量文档的情况下。

基本语法

基本语法如下：

```
GET /<index_name>/_search
{
 "query": {
    "wildcard": {
      "your_field_name": {
        "value": "your_search_pattern"
      }
    }
 }
}
```

示例

```
GET /employee/_search
{
  "query": {
    "wildcard": {
      "address.keyword": {
        "value": "*州*公园"
      }
    }
  }
}
```

regexp——正则匹配查询

regexp检索是一种基于正则表达式的检索方法。虽然该检索方式的功能强大，但建议在非必要情况下避免使用，以保持查询性能的高效和稳定。

基本语法

在Elasticsearch 8.x中，regexp 查询用于在字段中执行正则表达式匹配。这个查询可以用来搜索满足特定模式的文本，并且比 wildcard 查询更加灵活和强大。
基本语法如下：

```
GET /<index_name>/_search
{
 "query": {
    "regexp": {
      "your_field_name": {
        "value": "your_search_pattern"
      }
    }
 }
}
```

示例

```

GET /employee/_search
{
  "query": {
    "regexp": {
      "remark": {
        "value": "java.*"
      }
    }
  }
}
```

> .* 表示在java后可以跟随任意数量的任意字符

fuzzy——支持编辑距离的模糊查询

fuzzy检索是一种强大的搜索功能，它能够在用户输入内容存在拼写错误或上下文不一致时，仍然返回与搜索词相似的文档。通过使用编辑距离算法来度量输入词与文档中词条的相似程度，模糊查询在保证搜索结果相关性的同时，有效地提高了搜索容错能力。

> 编辑距离是指从一个单词转换到另一个单词需要编辑单字符的次数。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。

基本语法

基本语法如下：

```
GET /<index_name>/_search
{
  "query": {
    "fuzzy": {
      "your_field": {
        "value": "search_term",
        "fuzziness": "AUTO",
        "prefix_length": 1
      }
    }
  }
}


```

- fuzziness参数用于编辑距离的设置，其默认值为AUTO，支持的数值为[0，1，2]。如果值设置越界会报错。
- prefix_length: 搜索词的前缀长度，在此长度内不会应用模糊匹配。默认是0，即整个词都会被模糊匹配。

示例

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

term set——用于解决多值字段中的文档匹配问题

terms set检索是Elasticsearch中一种功能强大的检索类型，主要用于解决多值字段中的文档匹配问题，在处理具有多个属性、分类或标签的复杂数据时非常有用。

从应用场景来说，terms set检索在处理多值字段和特定匹配条件时具有很大的优势。它适用于标签系统、搜索引擎、电子商务系统、文档管理系统和技能匹配等场景。

基本语法

terms_set可以检索至少匹配一定数量给定词项的文档，其中匹配的数量可以是固定值，也可以是基于另一个字段的动态值

基本语法如下：

```
GET /<index_name>/_search
{
 "query": {
    "terms_set": {
      "<field_name>": {
        "terms": ["<term1>", "<term2>", ...],
        "minimum_should_match_field": "<minimum_should_match_field_name>" or
        "minimum_should_match_script": {
          "source": "<script>"
        }
      }
    }
 }
}


```

- <field_name>: 指定要查询的字段名，这个字段通常是一个多值字段。
- terms: 提供一组词项，用于在指定字段中进行匹配。
- minimum_should_match_field: 指定一个包含匹配数量的字段名，其值应用作要匹配的最少术语数，以便返回文档。
- minimum_should_match_script: 提供一个自定义脚本，用于动态计算匹配数量。如果需要动态设置匹配所需的术语数，这个参数将非常有用。

示例

假设我们有一个电影数据库，其中每部电影都有多个标签。现在，我们希望找到同时具有一定数量的给定标签的电影。

测试数据

```
PUT /movies
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text"
      },
      "tags": {
        "type": "keyword"
      },
      "tags_count": {
        "type": "integer"
      }
    }
  }
}

POST /movies/_bulk
{"index":{"_id":1}}
{"title":"电影1", "tags":["喜剧","动作","科幻"], "tags_count":3}
{"index":{"_id":2}}
{"title":"电影2", "tags":["喜剧","爱情","家庭"], "tags_count":3}
{"index":{"_id":3}}
{"title":"电影3", "tags":["动作","科幻","家庭"], "tags_count":3}
```

- 使用固定数量的term进行匹配

```
GET /movies/_search
{
  "query": {
    "terms_set": {
      "tags": {
        "terms": [
          "喜剧",
          "动作",
          "科幻"
        ],
        "minimum_should_match": 2
      }
    }
  }
}

GET /movies/_search
{
  "query": {
    "terms_set": {
      "tags": {
        "terms": [
          "喜剧",
          "动作",
          "科幻"
        ],
        "minimum_should_match_script": {
          "source": "2"
        }
      }
    }
  }
}
```

- 使用动态计算的term数量进行匹配

```
GET /movies/_search
{
  "query": {
    "terms_set": {
      "tags": {
        "terms": [
          "喜剧",
          "动作",
          "科幻"
        ],
        "minimum_should_match_field": "tags_count"
      }
    }
  }
}


GET /movies/_search
{
  "query": {
    "terms_set": {
      "tags": {
        "terms": [
          "喜剧",
          "动作",
          "科幻"
        ],
        "minimum_should_match_script": {
          "source": "doc['tags_count'].value*0.7"
        }
      }
    }
  }
}
```

1.3 全文检索

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。此类检索主要应用于非结构化文本数据，如文章和评论等。

match——分词查询

match查询是一种全文搜索查询，它使用分析器将查询字符串分解成单独的词条，并在倒排索引中搜索这些词条。match查询适用于文本字段，并且可以通过多种参数来调整搜索行为。

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.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。

3.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。

基本语法

一个基本的match查询的结构如下：

```
GET /<index_name>/_search
{
  "query": {
     "match": {
       "<field_name>": "<query_string>"
     }
  }
}

```

- <index_name> 是你要搜索的索引名称。
- <field_name> 是你要在其中搜索的字段名称。
- <query_string> 是你要搜索的文本字符串。

示例

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

# 分词后 and的效果
GET /employee/_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 /employee/_search
{
  "query": {
    "match": {
      "address": {
        "query": "广州公园",
        "minimum_should_match": 2
      }
    }
  }
}


```

multi_match——多字段查询

multi_match查询在Elasticsearch中用于在多个字段上执行相同的搜索操作。它可以接受一个查询字符串，并在指定的字段集合中搜索这个字符串。multi_match查询提供了灵活的匹配类型和操作符选项，以便根据不同的搜索需求调整搜索行为。

基本语法

一个基本的multi_match查询的结构如下：

```
GET /<index_name>/_search
{
 "query": {
    "multi_match": {
      "query": "<query_string>",
      "fields": ["<field1>", "<field2>", ...]
    }
 }
}

```

- <index_name> 是你要搜索的索引名称。
- <query_string> 是你要在多个字段中搜索的字符串。

- <field1>, <field2>, ... 是你要搜索的字段列表。

示例

```
GET /employee/_search
{
  "query": {
    "multi_match": {
      "query": "长沙java",
      "fields": [
        "address",
        "remark"
      ]
    }
  }
}
```

match_phrase短语查询

match_phrase查询在Elasticsearch中用于执行短语搜索，它不仅匹配整个短语，而且还考虑了短语中各个词的顺序和位置。这种查询类型对于搜索精确短语非常有用，尤其是在用户输入的查询与文档中的文本表达方式需要严格匹配时。

基本语法

一个基本的match_phrase查询的结构如下：

```
GET /<index_name>/_search
{
 "query": {
    "match_phrase": {
      "<field_name>": {
        "query": "<phrase>"
      }
    }
 }
}

```

- <index_name> 是你要搜索的索引名称。

- <field_name> 是你要在其中搜索短语的字段名称。
- <phrase> 是你要搜索的短语。

match_phrase查询还支持一个可选的slop参数，用于指定短语中词之间可以出现的最大位移数量。默认值为0，意味着短语中的词必须严格按照顺序出现。如果设置了非零的slop值，则允许短语中的某些词在一定范围内错位。

示例

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


GET /employee/_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
    }
  ]
}
```

```
39
```

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 /employee/_search
{
  "query": {
    "match_phrase": {
      "address": {
        "query": "广州云山",
        "slop": 2
      }
    }
  }
}


```

query_string——支持与或非表达式的查询

query_string查询是一种灵活的查询类型，它允许使用Lucene查询语法来构建复杂的搜索查询。这种查询类型支持多种逻辑运算符，包括与（AND）、或（OR）和非（NOT），以及通配符、模糊搜索和正则表达式等功能。query_string查询可以在单个或多个字段上进行搜索，并且可以处理复杂的查询逻辑。

应用场景包括高级搜索、数据分析和报表等，适合处理需满足特定需求、要求支持与或非表达式的复杂查询任务，通常用于专业领域或需要高级查询功能的应用中。

基本语法

query_string查询的基本语法结构如下：

```
GET /<index_name>/_search
{
 "query": {
    "query_string": {
      "query": "<your_query_string>",
      "default_field": "<field_name>"
    }
 }
}
```

- <your_query_string> 是查询逻辑，可以包含上述提到的逻辑运算符和通配符等
- <field_name> 是默认搜索字段，如果省略则会搜索所有可索引字段。

示例

- **未指定字段查询**

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

- **指定单个字段查询**

```
GET /employee/_search
{
  "query": {
    "query_string": {
      "default_field": "address",
      "query": "白云山 OR 橘子洲"
    }
  }
}
```

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

- **指定多个字段查询**

```
GET /employee/_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

在生产环境中推荐使用 simple_query_string 而不是 query_string 主要是因为 simple_query_string 提供了更宽松的语法，能够容忍一定程度的输入错误，而不会导致整个查询失败。

基本语法

simple_query_string 查询的基本语法结构通常如下所示：

```
GET /<index_name>/_search
{
 "query": {
    "simple_query_string": {
      "query": "<query_string>",
      "fields": ["<field1>", "<field2>", ...],
      "default_operator": "OR" // 或 "AND"
    }
 }
}
```

其中 <query_string> 是要搜索的查询表达式，<field1>, <field2>, ... 是搜索可以在其中进行的字段列表，default_operator 定义了查询字符串中未指定操作符时的默认逻辑运算符，可以是 "OR" 或 "AND"。

示例

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

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

小结

精确匹配与全文检索的本质区别主要表现在两个方面：

- 精确不对待检索文本进行分词处理，而是将整个文本视为一个完整的词条进行匹配。
- 全文检索则需要对文本进行分词处理。在分词后，每个词条将单独进行检索，并通过布尔逻辑（如与、或、非等）进行组合检索，以找到最相关的结果。

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": "yyyy-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

```
#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"
}

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": {
    "multi_match": {
      "fields": ["name","desc"],
      "query": "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": {}
    }
  }
}
```

- require_field_match: 该设置控制是否需要所有指定的高亮字段都匹配搜索查询，才能应用高亮。当设置为false时，只要任意一个字段匹配，该文档的匹配部分就会被高亮。如果设置为true，则所有指定的字段都必须匹配查询条件。

1.6 地理空间位置查询

地理空间位置查询是数据库和搜索系统中的一个重要特性，特别是在地理信息系统（GIS）和位置服务中。它允许用户基于地理位置信息来搜索和过滤数据。在Elasticsearch这样的全文搜索引擎中，地理空间位置查询被广泛应用，例如在旅行、房地产、物流和零售等行业，用于提供基于位置的搜索功能。

在Elasticsearch中，地理空间数据通常存储在geo_point字段类型中。这种字段类型可以存储纬度和经度坐标，用于表示地球上的一个点。
以下是一个使用geo_distance查询的例子，它会找到距离特定点一定距离内的所有文档。
1）确保索引中有一个geo_point字段，例如location。

```
PUT /my_index
{
 "mappings": {
    "properties": {
      "location": {
        "type": "geo_point"
      }
    }
 }
}
```

2）使用以下查询来找到距离给定坐标点（例如lat和lon）小于或等于10公里的所有文档：

```
GET /my_index/_search
{
 "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "geo_distance": {
          "distance": "10km",
          "distance_type": "arc",
          "location": {
            "lat": 39.9,
            "lon": 116.4
          }
        }
      }
    }
 }
}
```

在这个查询中：

- "bool" 是一个逻辑查询容器，用于组合多个查询子句。
- "match_all" 是一个匹配所有文档的查询子句。

- "geo_distance" 是一个地理距离查询，它允许您指定一个距离和一个点的坐标。
- "distance" 是查询的最大距离，单位可以是米(m)、公里(km)等。
- "distance_type" 可以是 arc（以地球表面的弧长为单位）或 plane（以直线距离为单位）。通常对于地球上的距离查询，建议使用 arc。
- "location" 是查询的参考点，包含纬度和经度坐标。

这个查询将返回索引|my_index中location字段在给定坐标点10公里范围内的所有文档。

示例

假设我们正在管理一个记录中国各地著名景点的索引，每个景点都带有地理坐标。以下是一些示例数据：

```
# 创建索引
PUT /tourist_spots
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "analyzer": "ik_max_word",
        "search_analyzer": "ik_max_word"
      },
      "location": {
        "type": "geo_point"
      }
    }
  }
}

# 插入文档
POST /tourist_spots/_doc
{
  "name": "故宫博物院",
  "location": {
    "lat": 39.9159,
    "lon": 116.3945
  },
  "city": "北京"
}

POST /tourist_spots/_doc
{
  "name": "西湖",
  "location": {
    "lat": 30.2614,
    "lon": 120.1479
  },
  "city": "杭州"
}

POST /tourist_spots/_doc
```

```
{
  "name": "雷峰塔",
  "location": {
    "lat": 30.2511,
    "lon": 120.1347
  },
  "city": "杭州"
}

POST /tourist_spots/_doc
{
  "name": "苏堤春晓",
  "location": {
    "lat": 30.2584,
    "lon": 120.1383
  },
  "city": "杭州"
}

# 搜索包含故宫或博物院的景点：
GET /tourist_spots/_search
{
  "query": {
    "match": {
      "name": "故宫 博物院"
    }
  }
}
# 查询北京附近的景点
GET /tourist_spots/_search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "geo_distance": {
          "distance": "10km",
          "distance_type": "arc",
```

```
          "location": {
            "lat": 39.9159,
            "lon": 116.3945
          }
        }
      }
    }
  }
}
# 查询杭州西湖5km附近的景点
#雷峰塔 - 位于西湖附近，距离约2.8公里。
#苏堤春晓 - 位于西湖边，距离西湖中心约1公里。
GET /tourist_spots/_search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "geo_distance": {
          "distance": "5km",
          "distance_type": "arc",
          "location": {
            "lat": 30.2614,
            "lon": 120.1479
          }
        }
      }
    }
  }
}


```

1.7 ElasticSearch8.x 向量检索

Elasticsearch 8.x 引入了一个重要的新特性：向量检索（Vector Search），特别是通过KNN（K-Nearest Neighbors）算法支持向量近邻检索。这一特性使得Elasticsearch在机器学习、数据分析和推荐系统等领域的应用变得更加广泛和强大。

向量检索的基本思路是，将文档（或数据项）表示为高维向量，并使用这些向量来执行相似性搜索。在Elasticsearch中，这些向量被存储在`dense_vector`类型的字段中，然后使用KNN算法来找到与给定向量最相似的其他向量。

向量检索示例1

```
PUT image-index
{
  "mappings": {
    "properties": {
      "image-vector": {
        "type": "dense_vector",
        "dims": 3
      },
      "title": {
        "type": "text"
      },
      "file-type": {
        "type": "keyword"
      },
      "my_label": {
        "type": "text"
      }
    }
  }
}

POST image-index/_bulk
{ "index": {} }
{ "image-vector": [-5, 9, -12], "title": "Image A", "file-type": "jpeg", "my_label": "red" }
{ "index": {} }
{ "image-vector": [10, -2, 3], "title": "Image B", "file-type": "png", "my_label": "blue" }
{ "index": {} }
{ "image-vector": [4, 0, -1], "title": "Image C", "file-type": "gif", "my_label": "red" }

POST image-index/_search
{
  "knn": {
    "field": "image-vector",
    "query_vector": [-5, 10, -12],
    "k": 10,
    "num_candidates": 100
  },
```

```
  "fields": [ "title", "file-type" ]
}
```

向量检索示例2

假设我们正在构建一个推荐系统，该系统基于用户对电影的评分向量来推荐相似电影。我们将使用Elasticsearch的向量检索功能来实现这一需求。

首先，我们需要创建一些测试数据，包括几部电影的评分向量。以下是一些示例数据：

```
# 创建索引
# 指定了rating_vector为5维的稠密向量，并启用了向量索引，同时选择了dot_product作为相似度计算方式。
PUT /movies
{
  "mappings": {
    "properties": {
      "rating_vector": {
        "type": "dense_vector",
        "dims": 5
      }
    }
  }
}
# 插入文档
# rating_vector字段存储了每部电影的评分向量，向量的维度为5。
POST /movies/_doc
{
  "title": "肖申克的救赎",
  "year": 1994,
  "genre": "犯罪",
  "rating_vector": [0.1, 0.3, 0.5, 0.7, 0.9]
}

POST /movies/_doc
{
  "title": "阿甘正传",
  "year": 1994,
  "genre": "剧情",
  "rating_vector": [0.2, 0.4, 0.6, 0.8, 1.0]
}

POST /movies/_doc
{
  "title": "泰坦尼克号",
  "year": 1997,
  "genre": "爱情",
  "rating_vector": [0.15, 0.35, 0.55, 0.75, 0.95]
}
```

测试用例1: 查询与《肖申克的救赎》评分向量最相似的电影

```
GET /movies/_search
{
  "knn": {
    "field": "rating_vector",
    "query_vector": [
      0.1,
      0.3,
      0.5,
      0.7,
      0.9
    ],
    "k": 1
  }
}
```

预期结果：返回与查询向量最相似的电影，应该是肖申克的救赎。

测试用例2: 查询与自定义向量[0.2, 0.4, 0.6, 0.8, 1.0]最相似的电影

```
GET /movies/_search
{
  "knn": {
    "field": "rating_vector",
    "query_vector": [
      0.2,
      0.4,
      0.6,
      0.8,
      1
    ],
    "k": 1
  }
}
```

预期结果：返回与查询向量最相似的电影，应该是阿甘正传。