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

1. ElasticSearch核心概念

1.1 搜索引擎基础知识

Elasticsearch功能的核心是搜索引擎，学习搜索引擎的基础知识对于加深Elasticsearch核心概念的理解大有裨益。

什么是全文检索

全文检索（Full-Text Search）是一种从大量文本数据中快速检索出包含指定词汇或短语的信息的技术。它允许用户输入一个或多个关键词，然后系统会在预先建立好的索引中查找包含这些关键词的文档或文档片段，并返回给用户。

全文检索广泛应用于各种信息管理系统和应用中，如搜索引擎、文档管理系统、电子邮件客户端、新闻聚合网站等。它可以帮助用户快速定位所需信息，提高检索效率和准确性。

> **查询**：有明确的搜索条件边界。比如，年龄 15~25 岁，颜色 = 红色，价格 < 3000，这里的 15、25、红色、3000 都是条件边界。即有明确的范围界定。
> **检索**：即全文检索，无搜索条件边界，召回结果取决于相关性，其相关性计算无明确边界性条件，如同义词、谐音、别名、错别字、混淆词、网络热梗等均可成为其相关性判断依据。

设想一个关于全文检索的场景，比如搜索Java设计模式：

id	标题	描述
1	Java中的23种设计模式	Java中23种设计模式，包括简单介绍,适用场景以及优缺点等
2	Java多线程设计模式	Java多线程与设计模式结合
3	设计模式之美	结合真实项目案例，从面向对象编程范式、设计原则、代码规范、重构技巧和设计模式5个方面详细介绍如何编写高质量代码。
4	JavaScript设计模式与开发实践	针对JavaScript语言特性全面介绍了更适合JavaScript程序员的了16个常用的设计模式
...	...	...
10亿	Java并发编程实战	深入浅出地介绍了Java线程和并发，是一本完美的Java并发参考手册
...	...	...

思考：用传统关系型数据库实现有什么问题？

如果是用MySQL存储文章，我们应该会使用这样的 SQL 去查询

```
select * from t_blog where content like "%Java设计模式%"
```

这种需要遍历所有的记录进行匹配，不但效率低，而且搜索结果不符合我们搜索时的期望。

全文检索实现原理

1）在全文检索中，首先需要对文本数据进行处理，包括分词、去除停用词等。然后，对处理后的文本数据建立索引，索引会记录每个单词在文档中的位置信息以及其他相关的元数据，如词频、权重等。这个过程通常使用倒排索引（inverted index）来实现，倒排索引将单词映射到包含该单词的文档列表中，以便快速定位相关文档。

2）当用户发起搜索请求时，搜索引擎会根据用户提供的关键词或短语，在建立好的索引中查找匹配的文档。搜索引擎会根据索引中的信息计算文档的相关性，并按照相关性排序返回搜索结果。用户可以通过不同的搜索策略和过滤条件来精确控制搜索结果的质量和范围。

什么是倒排索引

在一个文档集合中，每个文档都可视为一个词语的集合，倒排索引则是将词语映射到包含这个词语的文档的数据结构。

正排索引（Forward Index）和倒排索引（Inverted Index）是全文检索中常用的两种索引结构，它们在索引和搜索的过程中扮演不同的角色。

正排索引（正向索引）

正排索引是将文档按顺序排列并进行编号的索引结构。每个文档都包含了完整的文本内容，以及其他相关的属性或元数据，如标题、作者、发布日期等。在正排索引中，可以根据文档编号或其他属性快速定位和访问文档的内容。正排索引适合用于需要对文档进行整体检索和展示的场景，但对于包含大量文本内容的数据集来说，正排索引的存储和查询效率可能会受到限制。

在MySQL 中通过 ID 查找就是一种正排索引的应用。

倒排索引（反向索引）

倒排索引是根据单词或短语建立的索引结构。它将每个单词映射到包含该单词的文档列表中。倒排索引的建立过程是先对文档进行分词处理，然后记录每个单词在哪些文档中出现，以及出现的位置信息。通过倒排索引，可以根据关键词或短语快速找到包含这些词语的文档，并确定它们的相关性。倒排索引适用于在大规模文本数据中进行关键词搜索和相关性排序的场景，它能够快速定位文档，提高搜索效率。

我们在创建文章的时候，建立一个关键词与文章的对应关系表，就可以称之为倒排索引。如下图所示：

关键词	文章ID	是否命中索引
Java	1,2	√
设计模式	1,2,3,4	√
多线程	2	
JavaScript	4	

倒排索引的实现涉及到多个步骤：

1）文档预处理：对文档进行分词处理，移除停用词，并进行词干提取等操作。

2）构建词典：将处理后的词汇添加到词典中，并为每个词汇分配一个唯一的ID。

3) 创建倒排列表：对于词典中的每个词汇，创建一个倒排列表，记录该词汇在哪些文档中出现，以及出现的位置信息。

4）存储索引文件：将词典和倒排列表存储在磁盘上的索引文件中，通常会进行压缩处理以减小存储空间并提升查询效率。

5）查询处理：当用户发起搜索请求时，搜索引擎会从词典中查找每个关键词对应的倒排列表，并根据列表中的文档ID快速定位到包含这些关键词的文档。

1.2 ElasticSearch常用术语

我们可以对比MySQL来理解Elasticsearch，如下图所示。左侧是MySQL的基本概念，右侧是Elasticsearch对应的相似概念的定义。借由这种对比，我们可以更直观地看出Elasticsearch与传统数据库之间的关系及差异。

注意：在Elasticsearch 6.X之前的版本中，索引类似于SQL数据库，而type（类型）类似于表。然而，从ES 7.x版本开始，类型已经被弃用，一个索引只能包含一个文档类型。

索引

索引是Elasticsearch中用于存储和管理相关数据的逻辑容器。索引可以看作数据库中的一个表，它包含了一组具有相似结构的文档。在Elasticsearch中，数据以JSON格式的文档存储在索引内。每个索引具有唯一的名称，以便在执行搜索、更新和删除操作时进行引用。索引的名称可以由用户自定义，但必须全部小写。总之，索引是Elasticsearch中用于组织、存储和检索数据的一个核心概念。通过将数据划分为不同的索引，用户可以更有效地管理和查询相关数据。

映射

不少初学者对映射(Mapping)这个概念会感觉不好理解。映射类似于关系型数据库中的Schema，可以近似地理解为“表结构”。

映射的定义如下所示：

```
PUT /employee
{
  "mappings": {
    "properties": {
      "name": {
        "type": "keyword"
      },
      "sex": {
        "type": "integer"
      },
      "age": {
        "type": "integer"
      },
      "address": {
        "type": "text",
        "analyzer": "ik_max_word"
      },
      "remark": {
        "type": "text",
        "analyzer": "ik_smart"
      }
    }
  }
}
```

我们拿到一个业务需求后，往往会将业务细分会几个索引。每个索引都需要一个相对固定的表结构，包含但不限于字段名称、字段类型、是否需要分词、是否需要索引、是否需要存储、是否需要多字段类型等。这些都是设计映射时要考虑的问题。

文档

关系型数据库将数据以行或元组为单位存储在数据库表中，而Elasticsearch将数据以文档为单位存储在索引中。作为Elasticsearch的基本存储单元，文档是指存储在Elasticsearch索引中的JSON对象。文档中的数据由键值对构成。键是字段的名称，值是不同数据类型的字段。不同的数据类型包含但不限于字符串类型、数字类型、布尔类型、对象类型等。

```
{
  "_index": "employee",
  "_id": "2",
  "_version": 1,
  "_seq_no": 1,
  "_primary_term": 1,
  "found": true,
  "_source": {
    "name": "李四",
    "sex": 1,
    "age": 28,
    "address": "广州荔湾大厦",
    "remark": "java assistant"
  }
}
```

文档元数据，用于标注文档的相关信息：

- _index：文档所属的索引名
- _type：文档所属的类型名
- _id：文档唯一id
- _source: 文档的原始Json数据
- _version: 文档的版本号，修改删除操作_version都会自增1
- _seq_no: 和_version一样，一旦数据发生更改，数据也一直是累计的。Shard级别严格递增，保证后写入的Doc的_seq_no大于先写入的Doc的_seq_no。
- _primary_term: _primary_term主要是用来恢复数据时处理当多个文档的_seq_no一样时的冲突，避免Primary Shard上的写入被覆盖。每当Primary Shard发生重新分配时，比如重启，Primary选举等，_primary_term会递增1。

2. ElasticSearch索引操作详解

2.1 索引的实战场景

索引是具有相同结构的文档的集合，由唯一索引名称标定。一个集群中有多个索引，不同的索引代表不同的业务类型数据。下面列举一些应用索引的实战场景。

- 场景一：将采集的不同业务类型的数据存储到不同的索引
  a. 微博业务对应的索引weibo_index。
  b. 新闻业务对应的索引news_index。

c. 博客业务对应的索引blog_index。

以上3个索引包含的字段个数、字段名称、字段类型可能不完全一致。

- 场景二：按日期切分存储日志索引

a. 2024年7月的日志对应logs_202407。

b. 2024年8月的日志对应logs_202408。

以上logs_202407、logs_202408属于一类索引，只是考虑到日志新旧重要程度、数据量规模、索引分片大小和检索性能，按照时间维度进行了切分。

2.2 索引的基本操作

创建索引

创建索引的基本语法

创建索引的基本语法如下：

```
PUT /index_name
{
  "settings": {
    // 索引设置
  },
  "mappings": {
    "properties": {
      // 字段映射
    }
  }
}
```

必要的参数：

- **索引名称 (index_name)**

索引名称必须是小写字母，可以包含数字和下划线。

- **索引设置 (settings)**

1)分片数量 (number_of_shards)

一个索引的分片数决定了索引的并行度和数据分布。

示例：

```
"number_of_shards": 1


```

2)副本数量 (number_of_replicas)

副本提高了数据的可用性和容错能力。

示例：

```
"number_of_replicas": 1
```

- **映射 (mappings)**

字段属性 (properties)定义索引中文档的字段及其类型。常用字段类型包括：text, keyword, integer, float, date 等。

示例：

```
"properties": {
  "field1": {
    "type": "text"
  },
  "field2": {
    "type": "keyword"
  }
}

```

- 只定义索引名，而settings、mappings取默认值

```
#创建索引
PUT /myindex

#查看索引
GET /myindex


```

- 实践练习：创建一个名为 student_index 的索引，并设置一些自定义字段

创建一个名为 student_index 的索引，并设置以下字段：

- name（学生姓名）：text 类型
- age（年龄）：integer 类型
- enrolled_date(入学日期)：date 类型

```
PUT /student_index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "age": {
        "type": "integer"
      },
      "enrolled_date": {
        "type": "date"
      }
    }
  }
}
```

删除索引

查询操作可以分为两类：检索索引信息和搜索索引中的文档。

获取索引信息的基本语法如下：

```
GET /index_name


```

示例

```
# 获取名为 myindex的索引的信息：
GET myindex


```

搜索索引中的文档的基本语法如下：

```
GET /index_name/_search
{
  "query": {
    // 查询条件
  }
}


```

示例

```
# 搜索 name 字段包含 John 的文档
GET /student_index/_search
{
  "query": {
    "match": {
      "name": "John"
    }
  }
}
```

查询索引

查询操作可以分为两类：检索索引信息和搜索索引中的文档。

获取索引信息的基本语法如下：

```
GET /index_name
```

示例

```
# 获取名为 myindex的索引的信息：
GET myindex
```

搜索索引中的文档的基本语法如下：

```
GET /index_name/_search
{
  "query": {
    // 查询条件
  }
}


```

示例

```

# 搜索 name 字段包含 John 的文档
GET /student_index/_search
{
  "query": {
    "match": {
      "name": "John"
    }
  }
}


```

修改索引

动态更新索引的settings部分

更新索引设置

基本语法

```
PUT /index_name/_settings
{
  "index": {
    "setting_name": "setting_value"
  }
}


```

代码示例

将 student_index 的副本数量更新为 2：

```
PUT /student_index/_settings
{
  "index": {
    "number_of_replicas": 2
  }
}


```

动态更新索引的部分mapping字段信息

添加新的字段

基本语法

```
PUT /index_name/_mapping
{
  "properties": {
    "new_field": {
      "type": "field_type"
    }
  }
}


```

代码示例

向 student_index 添加一个名为 grade 的新字段，类型为 integer：

```
PUT /student_index/_mapping
{
  "properties": {
    "grade": {
      "type": "integer"
    }
  }
}
```

实践练习

向 student_index 添加一个名为 grade 的新字段，类型为 integer，并将副本数量更新为 2。

创建一个名为 student_index 的索引，并设置以下字段：

- name（学生姓名）：text 类型
- age（年龄）：integer 类型
- enrolled_date(入学日期)：date 类型

```
PUT /student_index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "age": {
        "type": "integer"
      },
      "enrolled_date": {
        "type": "date"
      }
    }
  }
}
```

2.3 索引别名详解

为什么需要别名

Elasitcsearch创建索引后，就不允许改索引名了。而在很多业务场景下，单一索引可能无法满足要求，举例如下。

- 场景1：面对PB级别的增量数据，对外提供服务的是基于日期切分的n个不同索引，每次检索都要指定数十个甚至数百个索引，非常麻烦。
- 场景2：线上提供服务的某个索引设计不合理，比如某字段分词定义不准确，那么如何保证对外提供服务不停止，也就是在不更改业务代码的前提下更换索引?

这两个真实业务场景问题都可以借助索引别名来解决。在很多实际业务场景中，使用别名会很方便、灵活、快捷，且使业务代码松耦合。

索引别名可以指向一个或多个索引，并且可以在任何需要索引名称的API中使用。别名提供了极大的灵活性，它允许用户执行以下操作。

- 在正在运行的集群上的一个索引和另一个索引之间进行透明切换。
- 对多个索引进行分组组合。例如last_three_months的索引别名就是对过去3个月的索引logstash_202303、logstash_202304、logstash_202305进行的组合。
- 在索引中的文档子集上创建“视图”，结合业务场景，缩小了检索范围，自然会提升检索效率。

如何为索引添加别名

- 创建索引的时候可以指定别名

```
PUT myindex
{
  "aliases": {
    "myindex_alias": {}
  },
  "settings": {
    "refresh_interval": "30s",
    "number_of_shards": 1,
    "number_of_replicas": 0
  }
}
```

- 为已有索引添加别名

要为现有索引添加别名，可以使用 _aliases API，基本语法如下：

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "index_name",
        "alias": "alias_name"
      }
    }
  ]
}
```

代码示例

```

#为 my_index 索引添加一个别名 my_index_alias:

POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "my_index",
        "alias": "my_index_alias"
      }
    }
  ]
}
```

多索引检索的实现方案

- 不使用别名的方案
  - 方式一：使用逗号对多个索引名称进行分隔

```
POST tlmall_logs_202401,tlmall_logs_202402,tlmall_logs_202403/_search
```

- 方式二：使用通配符进行多索引检索

```
POST tlmall_logs_*/_search

```

- 使用别名的方案

1) 使别名关联已有索引

示例

```
PUT tlmall_logs_202401
PUT tlmall_logs_202402
PUT tlmall_logs_202403

POST _aliases
{
  "actions": [
    {
      "add": {
        "index": "tlmall_logs_202401",
        "alias": "tlmall_logs_2024"
      }
    },
    {
      "add": {
        "index": "tlmall_logs_202402",
        "alias": "tlmall_logs_2024"
      }
    },
    {
      "add": {
        "index": "tlmall_logs_202403",
        "alias": "tlmall_logs_2024"
      }
    }
  ]
}
```

2) 使用别名进行检索

示例

```
1
2 POST tlmall_logs_2024/_search
3
4
5
```

思考：使用别名和基于索引的检索效率一样吗?

若索引和别名指向相同，则在相同检索条件下的检索效率是一致的，因为索引别名只是物理索引的软链接的名称而已。

注意：

1) 对相同索引别名的物理索引建议有一致的映射，以提升检索效率。

2) 推荐充分发挥索引别名在检索方面的优势，但在写入和更新时还得使用物理索引。

3. ElasticSearch文档操作详解

3.1 文档的介绍

作为Elasticsearch的基本存储单元，文档是指存储在Elasticsearch索引中的JSON对象。

3.2 文档的基本操作

新增文档

新增单个文档

基本语法

在ES8.x中，新增文档的操作可以通过POST或PUT请求完成，具体取决于是否指定了文档的唯一性标识（即ID）。如果在创建数据时指定了唯一性标识，可以使用POST或PUT请求；如果没有指定唯一性标识，只能使用POST请求。

使用POST请求新增文档

当不指定文档ID时，可以使用POST请求来新增文档，Elasticsearch会自动生成一个唯一的ID。语法如下：

```
POST /<index_name>/_doc
{
 "field1": "value1",
 "field2": "value2",
 // ... 其他字段
}


```

使用PUT请求新增文档

当指定了文档的唯一性标识（ID）时，可以使用PUT请求来新增或更新文档。如果指定的ID在索引中不存在，则会创建一个新文档；如果已存在，则会替换现有文档。语法如下：

```
PUT /<index_name>/_doc/<document_id>
{
 "field1": "value1",
 "field2": "value2",
 // ... 其他字段
}


```

PUT和POST的区别

在Elasticsearch 8.x中，PUT和POST请求在新增文档时的行为有所不同，主要体现在以下几个方面：

1. 指定文档ID：
   - PUT请求在创建或更新文档时必须指定文档的唯一ID。如果指定的ID已经存在，PUT请求会替换现有文档；如果不存在，则创建一个新文档。
   - POST请求在创建新文档时可以指定ID，也可以不指定。如果不指定ID，Elasticsearch会自动生成一个唯一的ID。
2. 幂等性：
   - PUT请求是幂等的，这意味着多次执行相同的PUT请求，即使是针对同一个文档，最终的结果都是一致的。
   - POST请求不是幂等的，多次执行相同的POST请求可能会导致创建多个文档。
3. 更新行为：
   - PUT请求在更新文档时会替换整个文档的内容，即使是文档中未更改的部分也会被新内容覆盖。
   - POST请求在更新文档时可以使用_update API，这样可以只更新文档中的特定字段，而不是替换整个文档。

示例

指定ID新增单个文档

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


```

不指定ID新增单条文档

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


```

批量新增文档

基本语法

在Elasticsearch 8.x中，批量新增文档可以通过_bulk API来实现。这个API允许您将多个索引、更新或删除操作组合成一个单一的请求，从而提高批量操作的效率。

以下是使用_bulk API的基本语法：

```
POST /<index_name>/_bulk
{ "index" : { "_index" : "<index_name>", "_id" : "<optional_document_id>" } }
{ "field1" : "value1", "field2" : "value2", ... }
{ "update" : { "_index" : "<index_name>", "_id" : "<document_id>" } }
{ "doc" : {"field1" : "new_value1", "field2" : "new_value2", ... }, "_op_type" : "update" }
{ "delete" : { "_index" : "<index_name>", "_id" : "<document_id>" } }
{ "index" : { "_index" : "<index_name>", "_id" : "<optional_document_id>" } }
{ "field1" : "value1", "field2" : "value2", ... }


```

每个操作都是一个独立的JSON对象，这些对象交替出现，形成一个请求体。每个index操作后面跟着的是要索引的文档内容，update操作包含了更新的文档内容和操作类型，而delete操作则直接指明要删除的文档ID。每个操作对象的开头都必须是index、update或delete，并且每个操作之间用一个空行分隔。

_bulk API支持哪些操作类型?

Elasticsearch的_bulk API支持以下四种操作类型：

- Index: 用于创建新文档或替换已有文档。
- Create: 如果文档不存在则创建，如果文档已存在则返回错误。
- Update: 用于更新现有文档。
- Delete: 用于删除指定的文档。

示例

Create: 如果文档不存在则创建，如果文档已存在则返回错误。

```
POST _bulk
{"create":{"_index":"article","_id":3}}
{"id":3,"title":"fox老师","content":"fox老师666","tags":["java","面向对象"],"create_time":1554015482530}
{"create":{"_index":"article","_id":4}}
{"id":4,"title":"mark老师","content":"mark老师NB","tags":["java","面向对象"],"create_time":1554015482530}


```

Index: 用于创建新文档或替换已有文档。

```
POST _bulk
{"index":{"_index":"article", "_id":3}}
{"id":3,"title":"图灵徐庶老师","content":"图灵学院徐庶老师666","tags":["java", "面向对象"],"create_time":1554015482530}
{"index":{"_index":"article",  "_id":4}}
{"id":4,"title":"图灵诸葛老师","content":"图灵学院诸葛老师NB","tags":["java", "面向对象"],"create_time":1554015482530}

```

实践练习：批量插入员工信息

1）创建员工索引

```
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"
          }
        }
      }
    }
  }
}
```

```
40
41
```

2）批量插入员工文档

```
1
2  POST /employee/_bulk
3  {"index":{"_index":"employee","_id":"1"}}
4  {"name":"张三","sex":1,"age":25,"address":"广州天河公园","remark":"java developer"}
5  {"index":{"_index":"employee","_id":"2"}}
6  {"name":"李四","sex":1,"age":28,"address":"广州荔湾大厦","remark":"java assistant"}
7  {"index":{"_index":"employee","_id":"3"}}
8  {"name":"王五","sex":0,"age":26,"address":"广州白云山公园","remark":"php developer"}
9  {"index":{"_index":"employee","_id":"4"}}
10 {"name":"赵六","sex":0,"age":22,"address":"长沙橘子洲","remark":"python assistant"}
11 {"index":{"_index":"employee","_id":"5"}}
12 {"name":"张龙","sex":0,"age":19,"address":"长沙麓谷企业广场","remark":"java architect
   assistant"}
13 {"index":{"_index":"employee","_id":"6"}}
14 {"name":"赵虎","sex":1,"age":32,"address":"长沙麓谷兴工国际产业园","remark":"java
   architect"}
15
16
17
```

查询文档

根据id查询文档

基本语法

在Elasticsearch 8.x中，根据文档的ID查询单个文档的标准语法是使用GET请求配合文档所在的索引名和文档ID。以下是具体的请求格式：

```
1 GET /<index_name>/_doc/<document_id>
2
3
```

在Elasticsearch 8.x中，使用Multi GET API可以根据ID查询多个文档。该API允许您在单个请求中指定多个文档的ID，并返回这些文档的信息。以下是Multi GET API的基本语法：

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


```

示例

根据id从employee索引中检索ID为1的单个文档

```
GET /employee/_doc/1


```

根据id列表从employee索引中批量检索多个文档

```
GET /employee/_mget
{
 "ids" : ["1", "2", "3"]
}


```

根据搜索关键词查询文档

基本语法

在Elasticsearch 8.x中，查询文档通常使用Query DSL（Domain Specific Language），这是一种基于JSON的语言，用于构建复杂的搜索查询。

```
GET /es_db/_search
{json请求体数据}


```

以下是一些常用的查询语法：

- 匹配所有文档

```
GET /<index_name>/_search
{
 "query": {
    "match_all": {}
 }
}

```

- 文本字段匹配

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

```

- 精确匹配（不分词）

```
GET /<index_name>/_search
{
 "query": {
    "term": {
      "<field_name>": {
        "value": "<exact_value>"
      }
    }
 }
}

```

- 范围查询

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

```

示例

```
#精确匹配, 姓名是张三的员工
GET /employee/_search
{
  "query": {
    "term": {
      "name": "张三"
    }
  }
}


# 全文检索,查询在广州白云山（搜索关键词）的员工
GET /employee/_search
{
  "query": {
    "match": {
      "address": "广州白云山"
    }
  }
}

#范围查询, 查询age在20至26岁之间的员工
GET /employee/_search
{
  "query": {
    "range": {
      "age": {
        "gte": 20,
        "lte": 26
      }
    }
  }
}


```

删除文档

删除单个文档

基本语法

在Elasticsearch 8.x中，删除单个文档的基本HTTP请求语法是：

```
DELETE /<index_name>/_doc/<document_id>


```

示例

删除员工id为1的文档

```
DELETE /employee/_doc/1


```

批量删除文档

基本语法

在Elasticsearch 8.x中，删除多个文档可以通过两种主要方法实现：

- 使用 _bulk API_bulk API允许您发送一系列操作请求，包括删除操作。每个删除请求是一个独立的JSON对象，格式如下：

```
POST /_bulk
{ "delete": {"_index": "{index_name}", "_id": "{id}"} }
{ "delete": {"_index": "{index_name}", "_id": "{id}"} }
{ "delete": {"_index": "{index_name}", "_id": "{id}"} }


```

- 使用 _delete_by_query API_delete_by_query API允许您根据查询条件删除文档。如果您想删除特定索引中匹配特定查询的所有文档，可以使用以下请求格式：

```
POST /{index_name}/_delete_by_query
{
 "query": {
    "<your_query>"
 }
}


```

示例

```

# 删除员工id为3和4的文档
POST _bulk
{"delete":{"_index":"employee","_id":3}}
{"delete":{"_index":"employee","_id":4}}


# 删除在广州的员工
POST /employee/_delete_by_query
{
 "query": {
    "match": {
      "address": "广州"
    }
 }
}


```

更新文档

更新单个文档

基本语法

在Elasticsearch 8.x版本中，更新操作通常通过_update接口执行，该接口允许您部分更新现有文档的字段。以下是更新文档的基本语法：

```
POST /{index_name}/_update/{id}
{
 "doc": {
    "<field>: <value>"
 }
}

```

示例

```
# 更新员工id为1的文档
POST /employee/_update/1
{
  "doc": {
    "age": 28
  }
}


```

批量更新文档

基本语法

在Elasticsearch 8.x中，更新多个文档可以通过两种主要方法实现：

- 使用 _bulk API

```
POST /_bulk
{ "update" : {"_index" : "<index_name>", "_id" : "<document_id>"} }
{ "doc" : {"field1" : "new_value1", "field2" : "new_value2"}, "upsert" : {"field1" : "new_value1", "field2" : "new_value2"} }
...


```

在这个请求中，每个update块代表一个更新操作，其中_index和_id指定了要更新的文档，doc部分包含了更新后的文档内容，upsert部分定义了如果文档不存在时应该插入的内容。

- 使用 _update_by_query API_update_by_query API允许您根据查询条件更新多个文档。这个操作是原子性的，意味着要么所有匹配的文档都被更新，要么一个都不会被更新。

```
POST /<index_name>/_update_by_query
{
 "query": {
    <!-- 定义更新文档的查询条件 -->
 },
 "script": {
    "source": "ctx._source.field = 'new_value'",
    "lang": "painless"
 }
}


```

在这个请求中，<index_name>是您要更新的索引名称，query部分定义了哪些文档需要被更新，script部分定义了如何更新这些文档的字段。

示例

```
# 更新员工id为3和4的文档
POST _bulk
{"update":{"_index":"employee","_id":3}}
{"doc":{"age":29}}
{"update":{"_index":"employee","_id":4}}
{"doc":{"age":27}}


#更新姓名为张三的员工
POST /employee/_update_by_query
{
  "query": {
    "term": {
      "name": "张三"
    }
  },
  "script": {
    "source": "ctx._source.age = 30"
  }
}


```

并发场景下更新文档如何保证线程安全

在Elasticsearch 7.x及以后的版本中，_seq_no和_primary_term取代了旧版本的_version字段，用于控制文档的版本。_seq_no代表文档在特定分片中的序列号，而_primary_term代表文档所在主分片的任期编号。这两个字段共同构成了文档的唯一版本标识符，用于实现乐观锁机制，确保在高并发环境下文档的一致性和正确更新。

当在高并发环境下使用乐观锁机制修改文档时，要带上当前文档的_seq_no和_primary_term进行更新：

```
POST /employee/_doc/1?if_seq_no=13&if_primary_term=1
{
  "name": "张三xxxx",
  "sex": 1,
  "age": 25
}
```

如果_seq_no和_primary_term不对，会抛出版本冲突异常：

```
{
  "error": {
    "root_cause": [
      {
        "type": "version_conflict_engine_exception",
        "reason": "[1]: version conflict, required seqNo [13], primary term [1]. current document has seqNo [14] and primary term [1]",
        "index_uuid": "7JwW1djNRKymS5P9FWgv7Q",
        "shard": "0",
        "index": "employee"
      }
    ],
    "type": "version_conflict_engine_exception",
    "reason": "[1]: version conflict, required seqNo [13], primary term [1]. current document has seqNo [14] and primary term [1]",
    "index_uuid": "7JwW1djNRKymS5P9FWgv7Q",
    "shard": "0",
    "index": "employee"
  },
  "status": 409
}
```

实践练习：实现某金融企业理财平台的理财产品信息检索功能

该企业的理财产品信息如下所示：

```
{
  "products":[
    {"productName":"理财产品A","annual_rate":"3.2200%","describe":"180天定期理财，最低20000起投，收益稳定，可以自助选择消息推送"}
    {"productName":"理财产品B","annual_rate":"3.1100%","describe":"90天定投产品，最低10000起投，每天收益到账消息推送"}
    {"productName":"理财产品C","annual_rate":"3.3500%","describe":"270天定投产品，最低40000起投，每天收益立即到账消息推送"}
    {"productName":"理财产品D","annual_rate":"3.1200%","describe":"90天定投产品，最低12000起投，每天收益到账消息推送"}
    {"productName":"理财产品E","annual_rate":"3.0100%","describe":"30天定投产品推荐，最低8000起投，每天收益会消息推送"}
    {"productName":"理财产品F","annual_rate":"2.7500%","describe":"热门短期产品，3天短期，无须任何手续费用，最低500起投，通过短信提示获取收益消息"}
  ]
}
```

1) 创建索引

创建一个名称为product_info的索引：

```
PUT /product_info
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
      "properties": {
        "productName": {
          "type": "text",
          "analyzer": "ik_smart"
        },
        "annual_rate":{
          "type":"keyword"
        },
        "describe": {
          "type": "text",
          "analyzer": "ik_smart"
        }
    }
  }
}
```

2) 新增文档

```
POST /product_info/_bulk
{"index":{}}
{"productName":"理财产品A","annual_rate":"3.2200%","describe":"180天定期理财，最低20000起投，收益稳定，可以自助选择消息推送"}
{"index":{}}
{"productName":"理财产品B","annual_rate":"3.1100%","describe":"90天定投产品，最低10000起投，每天收益到账消息推送"}
{"index":{}}
{"productName":"理财产品C","annual_rate":"3.3500%","describe":"270天定投产品，最低40000起投，每天收益立即到账消息推送"}
{"index":{}}
{"productName":"理财产品D","annual_rate":"3.1200%","describe":"90天定投产品，最低12000起投，每天收益到账消息推送"}
{"index":{}}
{"productName":"理财产品E","annual_rate":"3.0100%","describe":"30天定投产品推荐，最低8000起投，每天收益会消息推送"}
{"index":{}}
{"productName":"理财产品F","annual_rate":"2.7500%","describe":"热门短期产品，3天短期，无须任何手续费用，最低500起投，通过短信提示获取收益消息"}
```

3) 搜索数据

- 全文搜索

搜索描述内容包含每天收益到账消息推送的所有产品。

```
GET /product_info/_search
{
  "query": {
    "match": {
      "describe": "每天收益到账消息推送"
    }
  }
}
```

- 按查询条件搜索

搜索年化率在3.0000%到3.1300%之间的产品。

```
GET /product_info/_search
{
  "query": {
    "range": {
      "annual_rate": {
        "gte": "3.0000%",
        "lte": "3.1300%"
      }
    }
  }
}
```

4. ElasticSearch文件建模最佳实践

4.1 Elasticsearch中如何处理关联关系

Elasticsearch多表关联的问题是讨论最多的问题之一。多表关联通常指一对多或者多对多的数据关系，如博客及其评论的关系。

Elasticsearch并不擅长处理关联关系，一般会采用以下四种方法处理关联：

- 嵌套对象(Nested Object)

Nested类型适用于一对少量、子文档偶尔更新、查询频繁的场景。如果需要索引对象数组并保持数组中每个对象的独立性，则应使用Nested数据类型而不是Object数据类型。

Nested类型的优点是Nested文档可以将父子关系的两部分数据关联起来（例如博客与评论），可以基于Nested类型做任何查询。其缺点则是查询相对较慢，更新子文档时需要更新整篇文档。

- Join父子文档类型

Join类型用于在同一索引的文档中创建父子关系。Join类型适用于子文档数据量明显多于父文档的数据量的场景，该场景存在一对多量的关系，子文档更新频繁。举例来说，一个产品和供应商之间就是一对多的关联关系。当使用父子文档时，使用has_child或者has_parent做父子关联查询。

Join类型的优点是父子文档可独立更新。缺点则是维护Join关系需要占据部分内存，查询较Nested类型更耗资源。

- 宽表冗余存储

宽表适用于一对多或者多对多的关联关系。

宽表的优点是速度快。缺点则是索引更新或删除数据时，应用程序不得不处理宽表的冗余数据；并且由于冗余存储，某些搜索和聚合操作的结果可能不准确。

- 业务端关联

这是普遍使用的技术，即在应用接口层面处理关联关系。一般建议在存储层面使用两个独立索引存储，在实际业务层面这将分为两次请求来完成。

业务端关联适用于数据量少的多表关联业务场景。数据量少时，用户体验好；而数据量多时，两次查询耗时肯定会比较长，反而影响用户体验。

案例1： 博客作者信息变更

对象类型：

- 在每一博客的文档中都保留作者的信息
- 如果作者信息发生变化，需要修改相关的博客文档

```
DELETE blog
# 设置blog的 Mapping
PUT /blog
{
  "mappings": {
    "properties": {
      "content": {
        "type": "text"
      },
      "time": {
        "type": "date"
      },
      "user": {
        "properties": {
          "city": {
            "type": "text"
          },
          "userid": {
            "type": "long"
          },
          "username": {
            "type": "keyword"
          }
        }
      }
    }
  }
}

# 插入一条 blog信息
PUT /blog/_doc/1
{
  "content":"I like Elasticsearch",
  "time":"2022-01-01T00:00:00",
  "user":{
    "userid":1,
    "username":"Fox",
    "city":"Changsha"
  }
```

```
}


# 查询 blog信息
POST /blog/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"content": "Elasticsearch"}},
        {"match": {"user.username": "Fox"}}
      ]
    }
  }
}
```

案例2：包含对象数组的文档

```
DELETE /my_movies

# 电影的Mapping信息
PUT /my_movies
{
      "mappings" : {
      "properties" : {
        "actors" : {
          "properties" : {
            "first_name" : {
              "type" : "keyword"
            },
            "last_name" : {
              "type" : "keyword"
            }
          }
        },
        "title" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        }
      }
    }
}


# 写入一条电影信息
POST /my_movies/_doc/1
{
  "title":"Speed",
  "actors":[
    {
      "first_name":"Keanu",
      "last_name":"Reeves"
```

```
    },

    {
      "first_name":"Dennis",
      "last_name":"Hopper"
    }

  ]
}

# 查询电影信息
POST /my_movies/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"actors.first_name": "Keanu"}},
        {"match": {"actors.last_name": "Hopper"}}
      ]
    }
  }

}
```

思考：为什么会搜到不需要的结果?

存储时，内部对象的边界并没有考虑在内,JSON格式被处理成扁平式键值对的结构。当对多个字段进行查询时，导致了意外的搜索结果。可以用Nested Data Type解决这个问题。

```
"title":"Speed"
"actor".first_name: ["Keanu","Dennis"]
"actor".last_name: ["Reeves","Hopper"]

```

嵌套对象(Nested Object)

什么是Nested Data Type

- Nested数据类型: 允许对象数组中的对象被独立索引
- 使用nested 和properties 关键字，将所有actors索引到多个分隔的文档
- 在内部, Nested文档会被保存在两个Lucene文档中，在查询时做Join处理

```
DELETE /my_movies
# 创建 Nested 对象 Mapping
PUT /my_movies
{
      "mappings" : {
      "properties" : {
        "actors" : {
          "type": "nested",
          "properties" : {
            "first_name" : {"type" : "keyword"},
            "last_name" : {"type" : "keyword"}
          }},
        "title" : {
          "type" : "text",
          "fields" : {"keyword":{"type":"keyword","ignore_above":256}}
        }
      }
    }
}

POST /my_movies/_doc/1
{
  "title":"Speed",
  "actors":[
    {
      "first_name":"Keanu",
      "last_name":"Reeves"
    },

    {
      "first_name":"Dennis",
      "last_name":"Hopper"
    }

  ]
}

# Nested 查询
POST /my_movies/_search
```

```
{
  "query": {
    "bool": {
      "must": [
        {"match": {"title": "Speed"}},
        {
          "nested": {
            "path": "actors",
            "query": {
              "bool": {
                "must": [
                  {"match": {
                    "actors.first_name": "Keanu"
                  }},

                  {"match": {
                    "actors.last_name": "Hopper"
                  }}
                ]
              }
            }
          }
        }
      ]
    }
  }
}

# Nested Aggregation
POST /my_movies/_search
{
  "size": 0,
  "aggs": {
    "actors_agg": {
      "nested": {
        "path": "actors"
      },
      "aggs": {
        "actor_name": {
          "terms": {
```

```
            "field": "actors.first_name",
            "size": 10
          }
        }
      }
    }
  }
}


# 普通 aggregation不工作
POST /my_movies/_search
{
  "size": 0,
  "aggs": {
    "actors_agg": {
      "terms": {
        "field": "actors.first_name",
        "size": 10
      }
    }
  }
}
```

Join父子关联类型

- 对象和Nested对象的局限性: 每次更新，可能需要重新索引整个对象(包括根对象和嵌套对象)
- ES提供了类似关系型数据库中Join 的实现。使用Join数据类型实现，可以通过维护Parent/ Child的关系，从而分离两个对象
  - 父文档和子文档是两个独立的文档
  - 更新父文档无需重新索引子文档。子文档被添加，更新或者删除也不会影响到父文档和其他的子文档

设定 Parent/Child Mapping

```
DELETE /my_blogs

# 设定 Parent/Child Mapping
PUT /my_blogs
{
  "settings": {
    "number_of_shards": 2
  },
  "mappings": {
    "properties": {
      "blog_comments_relation": {
        "type": "join",
        "relations": {
          "blog": "comment"
        }
      },
      "content": {
        "type": "text"
      },
      "title": {
        "type": "keyword"
      }
    }
  }
}

```

索引父文档

```
#索引父文档
PUT /my_blogs/_doc/blog1
{
  "title":"Learning Elasticsearch",
  "content":"learning ELK ",
  "blog_comments_relation":{
    "name":"blog"
  }
}

#索引父文档
PUT /my_blogs/_doc/blog2
{
  "title":"Learning Hadoop",
  "content":"learning Hadoop",
  "blog_comments_relation":{
    "name":"blog"
  }
}
```

索引子文档

```
#索引子文档
PUT /my_blogs/_doc/comment1?routing=blog1
{
  "comment":"I am learning ELK",
  "username":"Jack",
  "blog_comments_relation":{
    "name":"comment",
    "parent":"blog1"
  }
}

#索引子文档
PUT /my_blogs/_doc/comment2?routing=blog2
{
  "comment":"I like Hadoop!!!!!",
  "username":"Jack",
  "blog_comments_relation":{
    "name":"comment",
    "parent":"blog2"
  }
}

#索引子文档
PUT /my_blogs/_doc/comment3?routing=blog2
{
  "comment":"Hello Hadoop",
  "username":"Bob",
  "blog_comments_relation":{
    "name":"comment",
    "parent":"blog2"
  }
}
```

注意：

- 父文档和子文档必须存在相同的分片上，能够确保查询join 的性能
- 当指定子文档时候，必须指定它的父文档Id。使用routing参数来保证，分配到相同的分片

查询

```
# 查询所有文档
POST /my_blogs/_search

#根据父文档ID查看
GET /my_blogs/_doc/blog2

# Parent Id 查询
POST /my_blogs/_search
{
  "query": {
    "parent_id": {
      "type": "comment",
      "id": "blog2"
    }
  }
}

# Has Child 查询,返回父文档
POST /my_blogs/_search
{
  "query": {
    "has_child": {
      "type": "comment",
      "query" : {
                "match": {
                    "username" : "Jack"
                }
            }
    }
  }
}


# Has Parent 查询，返回相关的子文档
POST /my_blogs/_search
{
  "query": {
    "has_parent": {
      "parent_type": "blog",
```

```
      "query" : {
                "match": {
                    "title" : "Learning Hadoop"
                }
            }
    }
  }
}

#通过ID ， 访问子文档
GET /my_blogs/_doc/comment3
#通过ID和routing ， 访问子文档
GET /my_blogs/_doc/comment3?routing=blog2

#更新子文档
PUT /my_blogs/_doc/comment3?routing=blog2
{
    "comment": "Hello Hadoop??",
    "blog_comments_relation": {
      "name": "comment",
      "parent": "blog2"
    }
}
```

多表关联方案对比

在Elasticsearch开发实战中，对于多表关联的设计要突破关系型数据库设计的思维定式。不建议在Elasticsearch中做多表关联操作，尽量在设计时使用扁平的宽表文档模型，或者尽量将业务转化为没有关联关系的文档形式，在文档建模处多下功夫，以提升检索效率。

	Nested嵌套类型	Join父子文档类型	宽表冗余存储	业务端关联
优点	文档存储在一起，读取性能高	父子文档可以独立更新，互不影响	以空间换时间	数据量少时，用户体验好
缺点	更新嵌套的子文档时，需要更新整个文档，查询相对较慢	Join关系的维护也耗费内存。读取性能Nested还差	字段冗余造成存储空间的浪费	数据量多，两次查询耗时比较长，影响用户体验
适用场景	对少量、子文档偶尔更新、查询频繁	子文档更新频繁	一对多或者多对多	数据量少

4.2 ElasticSearch文档建模的最佳实践

如何处理关联关系

- Object: 优先考虑反范式（Denormalization）
- Nested: 当数据包含多数值对象，同时有查询需求
- Child/Parent：关联文档更新非常频繁时

避免过多字段

- 一个文档中，最好避免大量的字段
  - 过多的字段数不容易维护
  - Mapping 信息保存在Cluster State 中，数据量过大，对集群性能会有影响
  - 删除或者修改数据需要reindex
- 默认最大字段数是1000，可以设置index.mapping.total_fields.limit限定最大字段数。·

思考：什么原因会导致文档中有成百上千的字段?

生产环境中，尽量不要打开 Dynamic，可以使用Strict控制新增字段的加入

- true： 未知字段会被自动加入，默认值
- false： 新字段不会被索引，但是会保存在_source
- strict： 新增字段不会被索引，文档写入失败

```
PUT /user
{
  "mappings": {
    "dynamic": "strict",
    "properties": {
      "name": {
        "type": "text"
      },
      "address": {
        "type": "object",
        "dynamic": "true"
      }
    }
  }
}
# 插入文档报错，原因为age为新增字段,会抛出异常
PUT /user/_doc/1
{
  "name":"fox",
  "age":32,
  "address":{
    "province":"湖南",
    "city":"长沙"
  }
}


```

对于多属性的字段，比如cookie，商品属性，可以考虑使用Nested

避免正则，通配符，前缀查询

正则，通配符查询，前缀查询属于Term查询，但是性能不够好。特别是将通配符放在开头，会导致性能的灾难

案例：针对版本号的搜索

```
# 将字符串转对象
PUT softwares/
{
  "mappings": {
    "properties": {
      "version": {
        "properties": {
          "display_name": {
            "type": "keyword"
          },
          "hot_fix": {
            "type": "byte"
          },
          "marjor": {
            "type": "byte"
          },
          "minor": {
            "type": "byte"
          }
        }
      }
    }
  }
}


#通过 Inner Object 写入多个文档
PUT softwares/_doc/1
{
  "version":{
  "display_name":"7.1.0",
  "marjor":7,
  "minor":1,
  "hot_fix":0
  }

}

PUT softwares/_doc/2
```

```
{
  "version":{
  "display_name":"7.2.0",
  "marjor":7,
  "minor":2,
  "hot_fix":0
  }
}

PUT softwares/_doc/3
{
  "version":{
  "display_name":"7.2.1",
  "marjor":7,
  "minor":2,
  "hot_fix":1
  }
}


# 通过 bool 查询,
POST softwares/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "match":{
            "version.marjor":7
          }
        },
        {
          "match":{
            "version.minor":2
          }
        }
      ]
    }
  }
}
```

```
80
```

避免空值引起的聚合不准

```
# Not Null 解决聚合的问题
DELETE /scores
PUT /scores
{
  "mappings": {
      "properties": {
        "score": {
          "type": "float",
          "null_value": 0
        }
      }
    }
}

PUT /scores/_doc/1
{
 "score": 100
}
PUT /scores/_doc/2
{
 "score": null
}

POST /scores/_search
{
  "size": 0,
  "aggs": {
    "avg": {
      "avg": {
        "field": "score"
      }
    }
  }
}
```

为索引的Mapping加入Meta 信息

- Mappings设置非常重要，需要从两个维度进行考虑

    - 功能：搜索，聚合，排序
    - 性能：存储的开销; 内存的开销; 搜索的性能
- Mappings设置是一个迭代的过程
    - 加入新的字段很容易（必要时需要update_by_query)
    - 更新删除字段不允许(需要Reindex重建数据)
    - 最好能对Mappings 加入Meta 信息，更好的进行版本管理
    - 可以考虑将Mapping文件上传git进行管理

```
PUT /my_index
{
  "mappings": {
    "_meta": {
      "index_version_mapping": "1.1"
    }
  }
}
```