

# day09-Elasticsearch02

在昨天的学习中，我们已经导入了大量数据到elasticsearch中，实现了商品数据的存储。不过查询商品数据时依然采用的是根据id查询，而非模糊搜索。

所以今天，我们来研究下elasticsearch的数据搜索功能。Elasticsearch提供了基于JSON的DSL（[Domain Specific Language](#)）语句来定义查询条件，其JavaAPI就是在组织DSL条件。

因此，我们先学习DSL的查询语法，然后再基于DSL来对照学习JavaAPI，就会事半功倍。

## 1.DSL查询

Elasticsearch的查询可以分为两大类：

- **叶子查询（Leaf query clauses）**：一般是在特定的字段里查询特定值，属于简单查询，很少单独使用。
- **复合查询（Compound query clauses）**：以逻辑方式组合多个叶子查询或者更改叶子查询的行为方式。

### 1.1.快速入门

我们依然在Kibana的DevTools中学习查询的DSL语法。首先来看查询的语法结构：

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "查询类型": {
5       // .. 查询条件
6     }
7   }
8 }
```

说明：

- `GET /{索引库名}/_search`：其中的 `_search` 是固定路径，不能修改

例如，我们以最简单的无条件查询为例，无条件查询的类型是：`match_all`，因此其查询语句如下：

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

由于match\_all无条件，所以条件位置不写即可。

执行结果如下：

JSON

took: 2 → 花费时间，单位是毫秒

timed\_out: false

\_shards

- total: 1
- successful: 1
- skipped: 0
- failed: 0

hits

- total
  - value: 10000 → 查询总条数 (超过10000条时最大只显示10000)
  - relation: gte
  - max\_score: 1
- hits: 10 → 命中的文档的数组
  - 0
    - \_index: items → 索引库
    - \_type: \_doc
    - \_id: 317578 → 文档id
    - \_score: 1
    - \_source
      - id: 317578
      - name: RIMOWA 21寸托运箱拉杆箱 SALSA AIR系列果绿色 820.70.36.4
      - price: 28900
      - stock: 10000
      - image: https://m.360buyimg.com/mobilecms/s720x720\_jfs/t6934/364/1195375010/84676/e9f2c55f/597ece38N0ddcbc77.jpg!q70.jpg.webp
      - category: 拉杆箱
      - brand: RIMOWA
      - sold: 0
      - commentCount: 0
      - isAD: false
      - updateTime: 1683342377000
  - 1
  - 2

原始文档

黑马程序员-研究院

你会发现虽然是match\_all，但是响应结果中并不会包含索引库中的所有文档，而是仅有10条。这是因为处于安全考虑，elasticsearch设置了默认的查询页数。

## 1.2.叶子查询

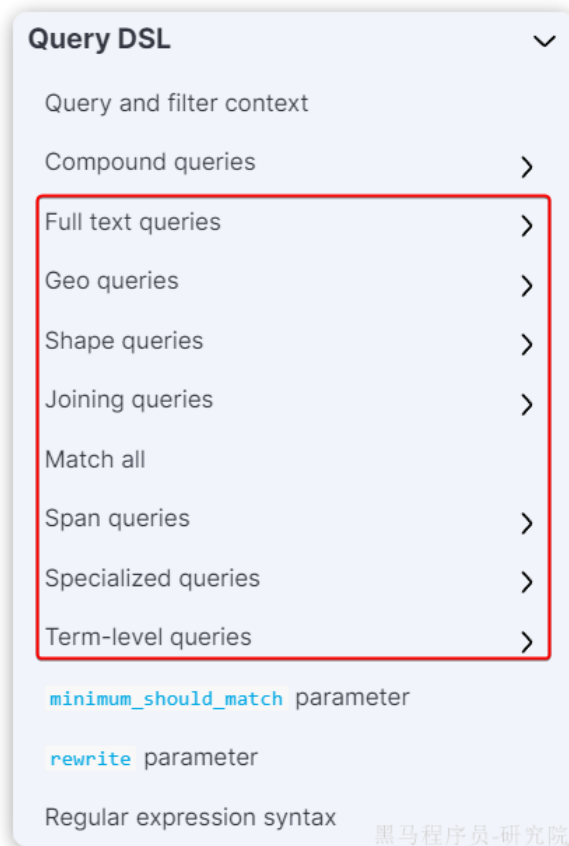
叶子查询的类型也可以做进一步细分，详情大家可以查看官方文档：

<https://www.elastic.co/guide/en/elasticsearch/reference/7.12/query-dsl.html>

## Query DSL | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Elasticsearch Guide [

如图：



这里列举一些常见的，例如：

- **全文检索查询（Full Text Queries）**：利用分词器对用户输入搜索条件先分词，得到词条，然后再利用倒排索引搜索词条。例如：
  - `match`：
  - `multi_match`
- **精确查询（Term-level queries）**：不对用户输入搜索条件分词，根据字段内容精确值匹配。但只能查找keyword、数值、日期、boolean类型的字段。例如：
  - `ids`
  - `term`
  - `range`
- **地理坐标查询**：用于搜索地理位置，搜索方式很多，例如：
  - `geo_bounding_box`：按矩形搜索

- `geo_distance` : 按点和半径搜索
- ...略

### 1.2.1.全文检索查询

全文检索的种类也很多，详情可以参考官方文档：

<https://www.elastic.co/guide/en/elasticsearch/reference/7.12/full-text-queries.html>

#### Full text queries | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Elasticsearch Guide [

以全文检索中的 `match` 为例，语法如下：

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "match": {
5       "字段名": "搜索条件"
6     }
7   }
8 }
```

示例：

The screenshot shows the Elasticsearch DevTools interface. The 'Console' tab is active, displaying a REST client request and its response. The request is a GET to `/items/_search` with a query object containing a match query for the field 'name' with the value '华为荣耀'. The response is a JSON object showing search results for the 'items' index, including a total count of 1 and a list of hits with details like id, score, and source.

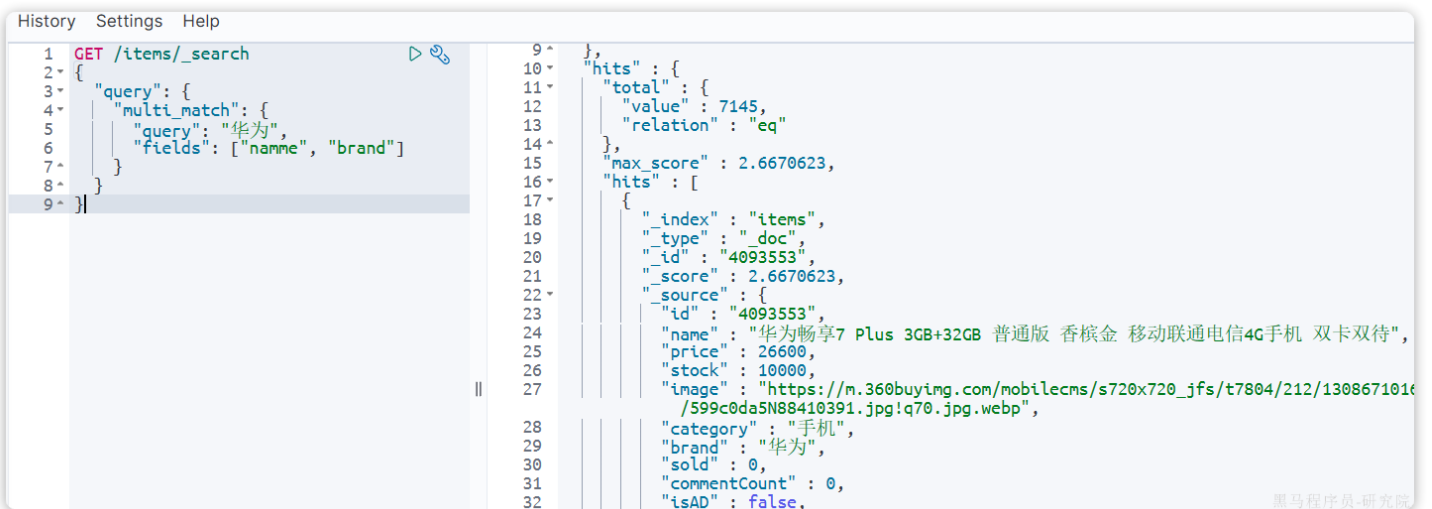
```
1 GET /items/_search
2 {
3   "query": {
4     "match": {
5       "name": "华为荣耀"
6     }
7   }
8 }
9
```

```
11 {
12   "total": {
13     "value": 1,
14     "relation": "eq"
15   },
16   "max_score": 9.623196,
17   "hits": [
18     {
19       "_index": "items",
20       "_type": "doc",
21       "_id": "39274582877",
22       "_score": 9.623196,
23       "_source": {
24         "id": "39274582877",
25         "name": "华为（HUAWEI） 荣耀V20 华为荣耀手机 幻夜黑 全网通(6G+128G)",
26         "price": 14800,
27         "stock": 10000,
28         "image": "https://m.360buyimg.com/mobilecms/s720x720_jfs/t1/22244/34/2984/a4b2fe1f88a94ca5.png!q70.jpg.webp",
29         "category": "手机",
30         "brand": "华为",
31         "sold": 0,
32       }
33     }
34   ]
35 }
```

与 `match` 类似的还有 `multi_match`，区别在于可以同时多个字段搜索，而且多个字段都要满足，语法示例：

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "multi_match": {
5       "query": "搜索条件",
6       "fields": ["字段1", "字段2"]
7     }
8   }
9 }
```

示例：



The screenshot shows a REST client interface with a query on the left and a JSON response on the right. The query is a `GET` request to `/items/_search` with a `multi_match` query for the term "华为" across the `name` and `brand` fields. The response is a JSON object containing search results.

```
1 GET /items/_search
2 {
3   "query": {
4     "multi_match": {
5       "query": "华为",
6       "fields": ["name", "brand"]
7     }
8   }
9 }
```

```
9  },
10 "hits": {
11   "total": {
12     "value": 7145,
13     "relation": "eq"
14   },
15   "max_score": 2.6670623,
16   "hits": [
17     {
18       "_index": "items",
19       "type": "doc",
20       "id": "4093553",
21       "score": 2.6670623,
22       "_source": {
23         "id": "4093553",
24         "name": "华为畅享7 Plus 3GB+32GB 普通版 香槟金 移动联通电信4G手机 双卡双待",
25         "price": 26600,
26         "stock": 10000,
27         "image": "https://n.360buyimg.com/mobilecms/s720x720_jfs/t7804/212/1308671016/599c0da5N88410391.jpg!q70.jpg.webp",
28         "category": "手机",
29         "brand": "华为",
30         "sold": 0,
31         "commentCount": 0,
32         "isAD": false,
```

### 1.2.2.精确查询

精确查询，英文是 `Term-level query`，顾名思义，词条级别的查询。也就是说不会对用户输入的搜索条件再分词，而是作为一个词条，与搜索的字段内容精确值匹配。因此推荐查找 `keyword`、数值、日期、`boolean` 类型的字段。例如：

- id
- price
- 城市
- 地名
- 人名

等等，作为一个整体才有含义的字段。

详情可以查看官方文档：

<https://www.elastic.co/guide/en/elasticsearch/reference/7.12/term-level-queries.html>

## Term-level queries | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Elasticsearch Guide [

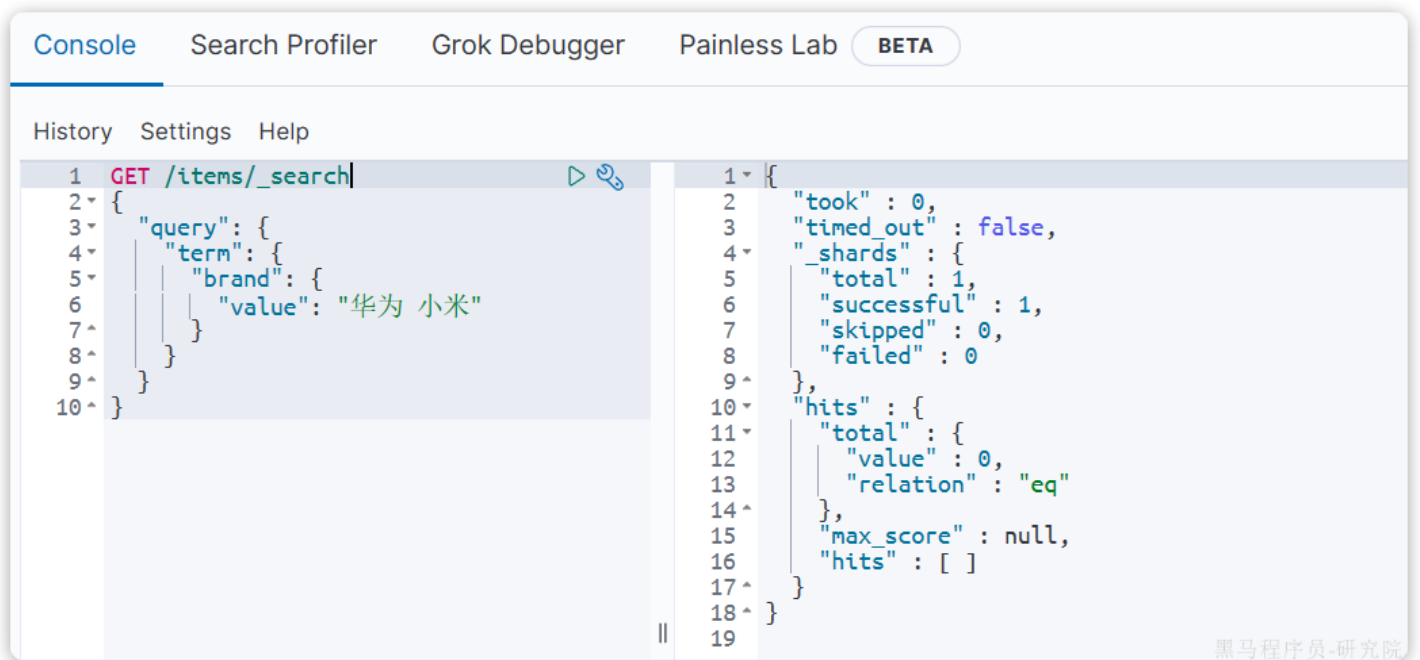
以 `term` 查询为例，其语法如下：

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "term": {
5       "字段名": {
6         "value": "搜索条件"
7       }
8     }
9   }
10 }
```

示例：

The screenshot shows the Elasticsearch DevTools interface. The left pane displays a REST client with a GET request to `/items/_search` containing a term query for the `brand` field with the value `"小米"`. The right pane shows the JSON response, which includes a `total` object with `value: 1498` and `relation: "eq"`, a `max_score` of `4.155314`, and a list of `hits`. The first hit is a document with `brand: "小米"`, which is highlighted with a red box. The document also contains fields like `name`, `price`, `stock`, `image`, `category`, `commentCount`, `isAD`, and `updateTime`.

当你输入的搜索条件不是词条，而是短语时，由于不做分词，你反而搜索不到：



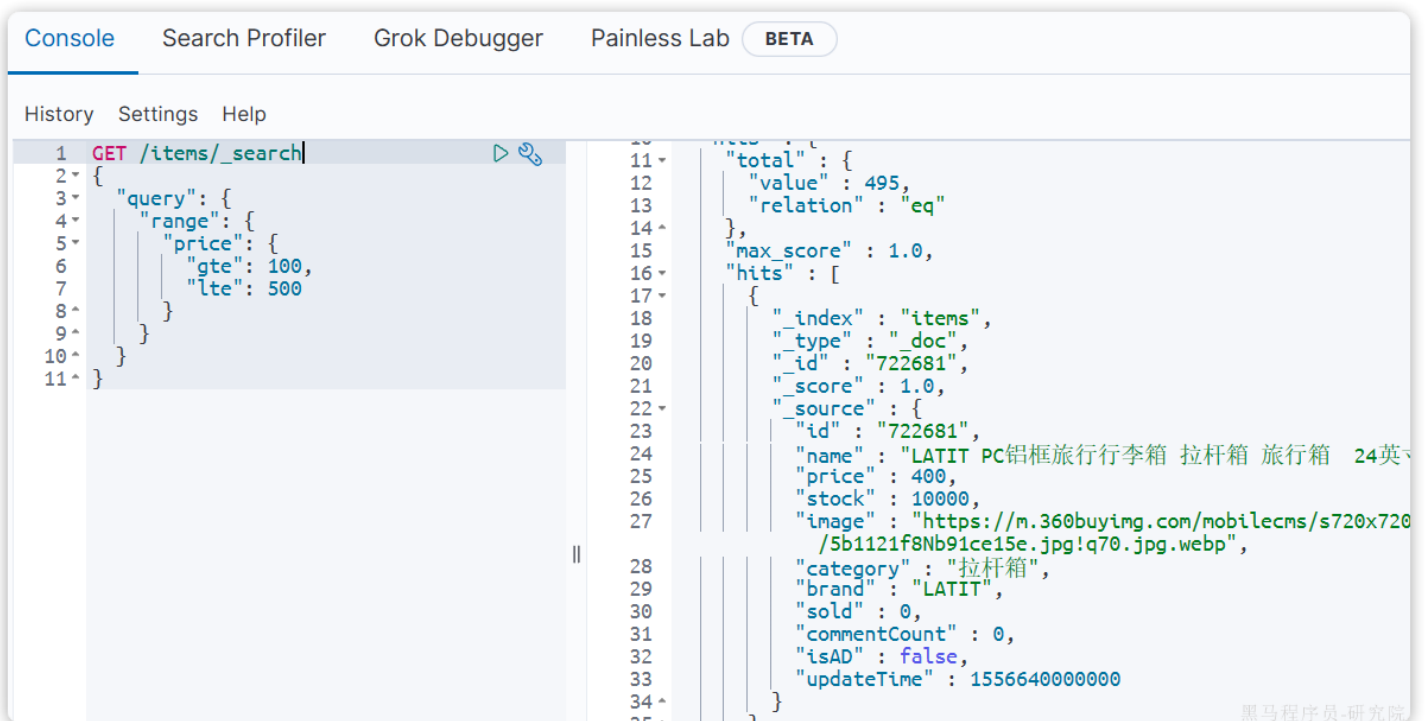
再来看下 `range` 查询，语法如下：

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "range": {
5       "字段名": {
6         "gte": {最小值},
7         "lte": {最大值}
8       }
9     }
10  }
11 }
```

`range` 是范围查询，对于范围筛选的关键字有：

- `gte`：大于等于
- `gt`：大于
- `lte`：小于等于
- `lt`：小于

示例：



## 1.3.复合查询

复合查询大致可以分为两类：

- 第一类：基于逻辑运算组合叶子查询，实现组合条件，例如
  - bool
- 第二类：基于某种算法修改查询时的文档相关性算分，从而改变文档排名。例如：
  - function\_score
  - dis\_max

其它复合查询及相关语法可以参考官方文档：

<https://www.elastic.co/guide/en/elasticsearch/reference/7.12/compound-queries.html>

### Compound queries | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Elasticsearch Guide [

### 1.3.1.算分函数查询（选讲）

当我们利用match查询时，文档结果会根据与搜索词条的**关联度打分**（**\_score**），返回结果时按照分值降序排列。

例如，我们搜索 "手机"，结果如下：



```
History Settings Help
1 GET /items/_search
2 {
3   "query": {
4     "match": {
5       "name": "4G手机"
6     }
7   }
8 }

15 "max_score" : 16.338163,
16 "hits" : [
17   {
18     "_index" : "items",
19     "_type" : "doc",
20     "id" : "15060942950",
21     "score" : 16.338163,
22     "source" : {
23       "id" : "15060942950",
24       "name" : "华为（HUAWEI） 荣耀8 全网通4G手机 双卡双
25       "price" : 72900,
26       "stock" : 10000,
27       "image" : "https://m.360buyimg.com/mobilecms/s720x
28       /598bdd54Nc57caca1.jpg!q70.jpg.webp",
29       "category" : "手机",
30       "brand" : "华为",
31       "sold" : 0,
32       "commentCount" : 0,
33       "isAD" : false,
34       "updateTime" : 1556640000000
35     }
36   },
37   {
38     "_index" : "items",
39     "_type" : "doc",
40     "id" : "15060942951",
41     "score" : 16.338163,
42     "source" : {
43       "id" : "15060942951",
44       "name" : "华为（HUAWEI） 荣耀8 全网通4G手机 双卡双
45       "price" : 7400,
46       "stock" : 10000,
47       "image" : "https://m.360buyimg.com/mobilecms/s720x
48       /598bdd54Nc57caca1.jpg!q70.jpg.webp",
49       "category" : "手机",
50       "brand" : "华为",
51       "sold" : 0,
52     }
53   }
54 ]
```

黑马程序员-研究院

从elasticsearch5.1开始，采用的相关性打分算法是BM25算法，公式如下：

## BM25算法

$$\text{Score}(Q,d) = \sum_i^n \log \left( 1 + \frac{N-n+0.5}{n+0.5} \right) \cdot \frac{f_i}{f_i + k_1 \cdot (1 - b + b \cdot \frac{dl}{avgdl})}$$

基于这套公式，就可以判断出某个文档与用户搜索的关键字之间的关联度，还是比较准确的。但是，在实际业务需求中，常常会有竞价排名的功能。不是相关度越高排名越靠前，而是掏的钱多的排名靠前。

例如在百度中搜索Java培训，排名靠前的就是广告推广：



要想认为控制相关性算分，就需要利用elasticsearch中的function score 查询了。

### 基本语法：

function score 查询中包含四部分内容：

- **原始查询条件**：query部分，基于这个条件搜索文档，并且基于BM25算法给文档打分，**原始算分**（query score）
- **过滤条件**：filter部分，符合该条件的文档才会重新算分
- **算分函数**：符合filter条件的文档要根据这个函数做运算，得到的**函数算分**（function score），有四种函数
  - weight：函数结果是常量
  - field\_value\_factor：以文档中的某个字段值作为函数结果
  - random\_score：以随机数作为函数结果
  - script\_score：自定义算分函数算法
- **运算模式**：算分函数的结果、原始查询的相关性算分，两者之间的运算方式，包括：
  - multiply：相乘
  - replace：用function score替换query score
  - 其它，例如：sum、avg、max、min

function score的运行流程如下：

- 1) 根据**原始条件**查询搜索文档，并且计算相关性算分，称为**原始算分**（query score）
- 2) 根据**过滤条件**，过滤文档
- 3) 符合**过滤条件**的文档，基于**算分函数**运算，得到**函数算分**（function score）

- 4) 将**原始算分** (query score) 和**函数算分** (function score) 基于**运算模式**做运算，得到最终结果，作为相关性算分。

因此，其中的关键点是：

- 过滤条件：决定哪些文档的算分被修改
- 算分函数：决定函数算分的算法
- 运算模式：决定最终算分结果

示例：给iPhone这个品牌的手机算分提高十倍，分析如下：

- 过滤条件：品牌必须为iPhone
- 算分函数：常量weight，值为10
- 算分模式：相乘multiply

对应代码如下：

```
1 GET /hotel/_search
2 {
3   "query": {
4     "function_score": {
5       "query": { .... }, // 原始查询，可以是任意条件
6       "functions": [ // 算分函数
7         {
8           "filter": { // 满足的条件，品牌必须是Iphone
9             "term": {
10              "brand": "Iphone"
11            }
12          },
13          "weight": 10 // 算分权重为2
14        }
15      ],
16      "boost_mode": "multiply" // 加权模式，求乘积
17    }
18  }
19 }
```

### 1.3.2.bool查询

bool查询，即布尔查询。就是利用逻辑运算来组合一个或多个查询子句的组合。bool查询支持的逻辑运算有：

- must：必须匹配每个子查询，类似“与”
- should：选择性匹配子查询，类似“或”
- must\_not：必须不匹配，**不参与算分**，类似“非”
- filter：必须匹配，**不参与算分**

bool查询的语法如下：

```
1 GET /items/_search
2 {
3   "query": {
4     "bool": {
5       "must": [
6         {"match": {"name": "手机"}}
7       ],
8       "should": [
9         {"term": {"brand": {"value": "vivo" }}},
10        {"term": {"brand": {"value": "小米" }}}
11      ],
12       "must_not": [
13         {"range": {"price": {"gte": 2500}}}
14      ],
15       "filter": [
16         {"range": {"price": {"lte": 1000}}}
17      ]
18     }
19   }
20 }
```

出于性能考虑，与搜索关键字无关的查询尽量采用must\_not或filter逻辑运算，避免参与相关性算分。

例如黑马商城的搜索页面：

黑马欢迎您! 请登录 免费注册

首页 | 我的购物车 | 我的黑马 | 黑马会员 | 企业采购

 黑马商城  
HMALL.COM

搜索关键字

搜索

全部结果:

过滤条件

分类	手机	曲面电视	拉杆箱	休闲鞋	休闲鞋	硬盘	真皮包			
品牌	希捷	小米	华为	oppo	尤妮佳	莎米特	意尔康	新秀丽	Apple	锤子
价格	100以下	100~299元	300~599元	600~899元	900~1599元	1600以上元				

黑马程序员-研九院

其中输入框的搜索条件肯定要参与相关性算分，可以采用match。但是价格范围过滤、品牌过滤、分类过滤等尽量采用filter，不要参与相关性算分。

比如，我们要搜索 手机 ，但品牌必须是 华为 ，价格必须是 900~1599 ，那么可以这样写：

```
1 GET /items/_search
2 {
3   "query": {
4     "bool": {
5       "must": [
6         {"match": {"name": "手机"}}
7       ],
8       "filter": [
9         {"term": {"brand": { "value": "华为" }}}},
10        {"range": {"price": {"gte": 90000, "lt": 159900}}}
11      ]
12    }
13  }
14 }
```

## 1.4.排序

elasticsearch默认是根据相关度算分（`_score`）来排序，但是也支持自定义方式对搜索结果排序。不过分词字段无法排序，能参与排序字段类型有：`keyword` 类型、数值类型、地理坐标类型、日期类型等。

详细说明可以参考官方文档：

<https://www.elastic.co/guide/en/elasticsearch/reference/7.12/sort-search-results.html>

### Sort search results | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs > Elasticsearch Guide [

语法说明：

```
1 GET /indexName/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "sort": [
7     {
8       "排序字段": {
9         "order": "排序方式asc和desc"
10      }
11    ]
12  }
```

示例，我们按照商品价格排序：

```
1 GET /items/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "sort": [
7     {
8       "price": {
9         "order": "desc"
10      }
11    ]
12  }
```

## 1.5.分页

elasticsearch 默认情况下只返回top10的数据。而如果要查询更多数据就需要修改分页参数了。

### 1.5.1.基础分页

elasticsearch中通过修改 `from`、`size` 参数来控制要返回的分页结果：

- `from`：从第几个文档开始
- `size`：总共查询几个文档

类似于mysql中的 `limit ?, ?`

官方文档如下：

<https://www.elastic.co/guide/en/elasticsearch/reference/7.12/paginate-search-results.html>

### Paginate search results | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Elasticsearch Guide [

语法如下：

```
1 GET /items/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "from": 0, // 分页开始的位置，默认为0
7   "size": 10, // 每页文档数量，默认10
8   "sort": [
9     {
10      "price": {
11        "order": "desc"
12      }
13    }
14  ]
15 }
```

## 1.5.2.深度分页

elasticsearch的数据一般会采用分片存储，也就是把一个索引中的数据分成N份，存储到不同节点上。这种存储方式比较有利于数据扩展，但给分页带来了一些麻烦。

比如一个索引库中有100000条数据，分别存储到4个分片，每个分片25000条数据。现在每页查询10条，查询第99页。那么分页查询的条件如下：

```
1 GET /items/_search
2 {
3   "from": 990, // 从第990条开始查询
4   "size": 10, // 每页查询10条
```

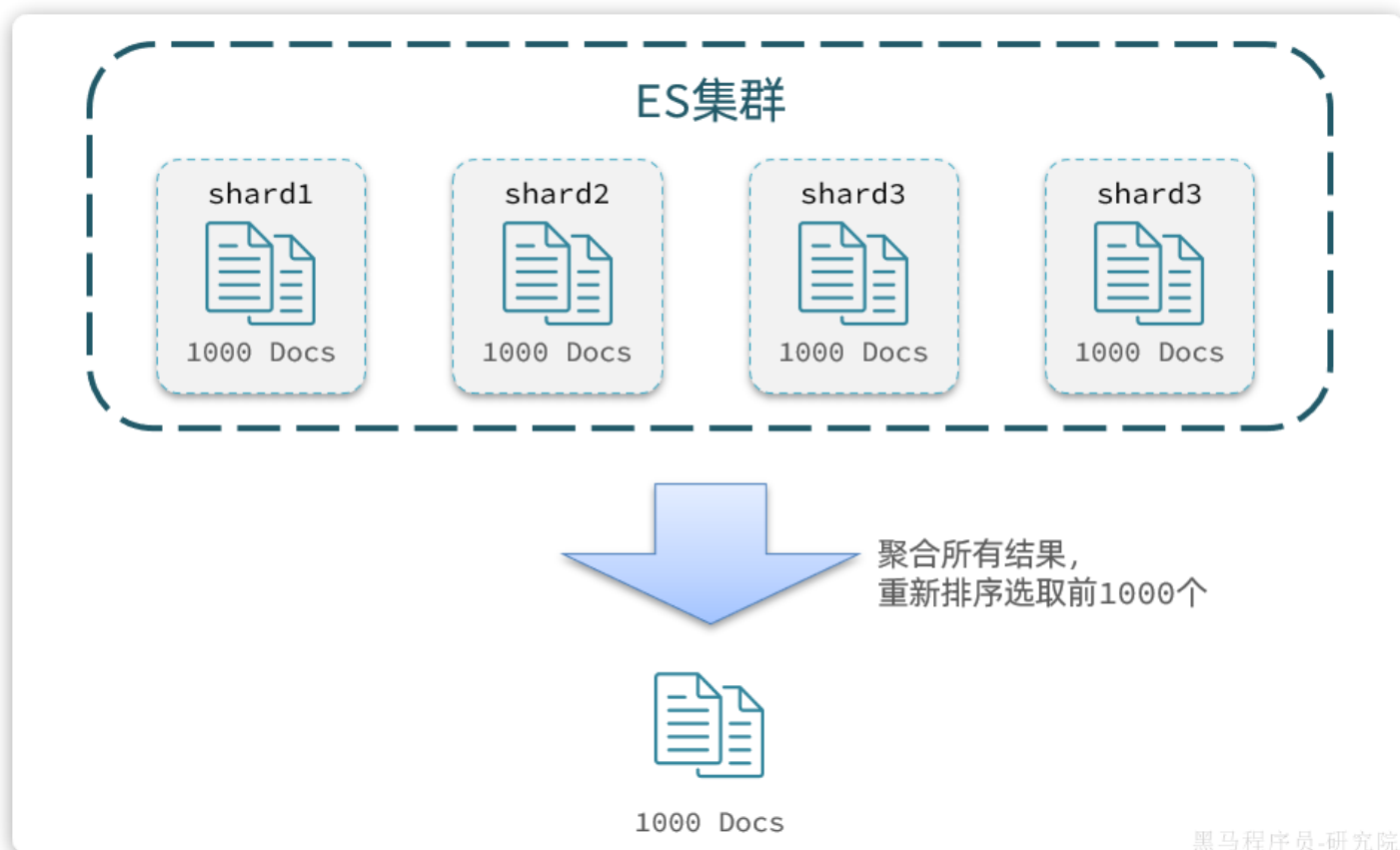
```
5  "sort": [  
6    {  
7      "price": "asc"  
8    }  
9  ]  
10 }
```

从语句来分析，要查询第990~1000名的数据。

从实现思路来分析，肯定是将所有数据排序，找出前1000名，截取其中的990~1000的部分。但问题来了，我们如何才能找到所有数据中的前1000名呢？

要知道每一片的数据都不一样，第1片上的第900~1000，在另1个节点上并不一定依然是900~1000名。所以我们只能在每一个分片上都找出排名前1000的数据，然后汇总到一起，重新排序，才能找出整个索引库中真正的前1000名，此时截取990~1000的数据即可。

如图：



试想一下，假如我们现在要查询的是第999页数据呢，是不是要找第9990~10000的数据，那岂不是需要把每个分片中的前10000名数据都查询出来，汇总在一起，在内存中排序？如果查询的分页深度更深呢，需要一次检索的数据岂不是更多？

由此可知，当查询分页深度较大时，汇总数据过多，对内存和CPU会产生非常大的压力。

因此elasticsearch会禁止 `from+ size` 超过10000的请求。



针对深度分页，elasticsearch提供了两种解决方案：

- `search after`：分页时需要排序，原理是从上一次的排序值开始，查询下一页数据。官方推荐使用的方式。
- `scroll`：原理将排序后的文档id形成快照，保存下来，基于快照做分页。官方已经不推荐使用。

详情见文档：

<https://www.elastic.co/guide/en/elasticsearch/reference/7.12/paginate-search-results.html>

### Paginate search results | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Elasticsearch Guide [

#### ！ 总结：

大多数情况下，我们采用普通分页就可以了。查看百度、京东等网站，会发现其分页都有限制。例如百度最多支持77页，每页不足20条。京东最多100页，每页最多60条。

因此，一般我们采用限制分页深度的方式即可，无需实现深度分页。

## 1.6.高亮

### 1.6.1.高亮原理

什么是高亮显示呢？

我们在百度，京东搜索时，关键字会变成红色，比较醒目，这叫高亮显示：



```

1 GET /{索引库名}/_search
2 {
3   "query": {
4     "match": {
5       "搜索字段": "搜索关键字"
6     }
7   },
8   "highlight": {
9     "fields": {
10      "高亮字段名称": {
11        "pre_tags": "<em>",
12        "post_tags": "</em>"
13      }
14    }
15  }
16 }

```

## ! 注意:

- 搜索必须有查询条件，而且是全文检索类型的查询条件，例如 `match`
- 参与高亮的字段必须是 `text` 类型的字段
- 默认情况下参与高亮的字段要与搜索字段一致，除非添加：  
`required_field_match=false`

## 示例:

The screenshot shows a REST client interface with the following details:

- Console Tab:** Shows the request and response.
  - Request (Line 1-16):**

```

1 GET /items/_search
2 {
3   "query": {
4     "match": {
5       "name": "脱脂牛奶"
6     }
7   },
8   "highlight": {
9     "fields": {
10      "name": {
11        "pre_tags": "<em>",
12        "post_tags": "</em>"
13      }
14    }
15  }
16 }

```
  - Response (Line 15-38):**

```

15 "max_score" : 16.19757,
16 "hits" : [
17   {
18     "_index" : "items",
19     "_type" : "doc",
20     "_id" : "33449279171",
21     "_score" : 16.19757,
22     "_source" : {
23       "id" : "33449279171",
24       "name" : "意大利 进口牛奶 葛兰纳诺脱脂纯牛奶 成人牛奶 进口脱脂纯牛奶1Lx6盒",
25       "price" : 3500,
26       "stock" : 10000,
27       "image" : "https://m.360buyimg.com/mobilecms/s720x720_jfs/t1/25045/9/2656/164517/5c20699dE9b7f4c9/1a05e9bdd2c5d59e.jpg!q70.jpg.webp",
28       "category" : "牛奶",
29       "brand" : "葛兰纳诺",
30       "sold" : 0,
31       "commentCount" : 0,
32       "isAD" : false,
33       "updateTime" : 1556640000000
34     },
35     "highlight" : {
36       "name" : [
37         "意大利 进口<em>牛奶</em> 葛兰纳诺<em>脱脂</em>纯<em>牛奶</em> 成人<em>牛奶</em> 进口<em>脱脂</em>纯<em>牛奶</em>1Lx6盒"
38       ]
39     }
40   }
41 ]

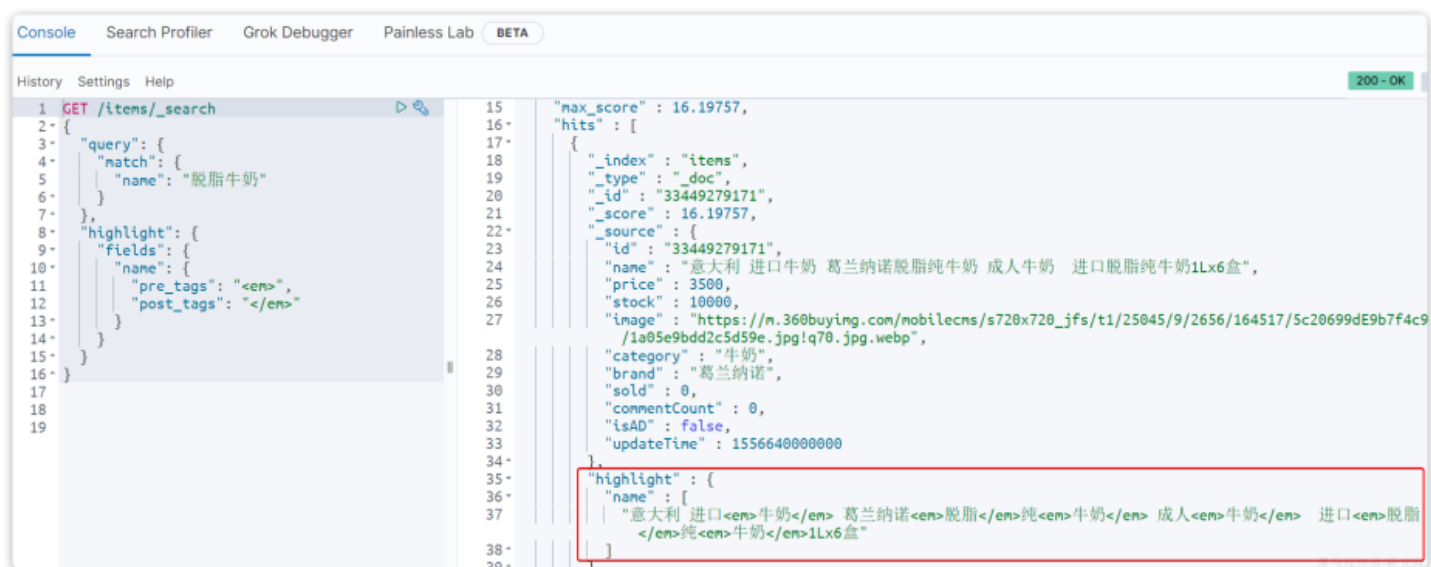
```

## 1.7.总结

查询的DSL是一个大的JSON对象，包含下列属性：

- `query`：查询条件
- `from` 和 `size`：分页条件
- `sort`：排序条件
- `highlight`：高亮条件

示例：



```
1 GET /items/_search
2 {
3   "query": {
4     "match": {
5       "name": "脱脂牛奶"
6     }
7   },
8   "highlight": {
9     "fields": {
10      "name": {
11        "pre_tags": "<em>",
12        "post_tags": "</em>"
13      }
14    }
15  }
16 }
17
18 "max_score" : 16.19757,
19 "hits" : [
20   {
21     "_index" : "items",
22     "_type" : "_doc",
23     "_id" : "33449279171",
24     "_score" : 16.19757,
25     "_source" : {
26       "id" : "33449279171",
27       "name" : "意大利 进口牛奶 葛兰纳诺脱脂纯牛奶 成人牛奶 进口脱脂纯牛奶1Lx6盒",
28       "price" : 3500,
29       "stock" : 10000,
30       "image" : "https://n.360buyimg.com/mobilecms/s720x720_jfs/t1/25045/9/2656/164517/5c20699dE9b7f4c9/1a05e9bdd2c5d59e.jpg!q70.jpg.webp",
31       "category" : "牛奶",
32       "brand" : "葛兰纳诺",
33       "sold" : 0,
34       "commentCount" : 0,
35       "isAD" : false,
36       "updateTime" : 1556640000000
37     },
38     "highlight" : {
39       "name" : [
40         "意大利 进口<em>牛奶</em> 葛兰纳诺<em>脱脂</em>纯<em>牛奶</em> 成人<em>牛奶</em> 进口<em>脱脂</em>纯<em>牛奶</em>1Lx6盒"
41       ]
42     }
43   }
44 ]
```

## 2.RestClient查询

文档的查询依然使用昨天学习的 `RestHighLevelClient` 对象，查询的基本步骤如下：

- 1) 创建 `request` 对象，这次是搜索，所以是 `SearchRequest`
- 2) 准备请求参数，也就是查询DSL对应的JSON参数
- 3) 发起请求
- 4) 解析响应，响应结果相对复杂，需要逐层解析

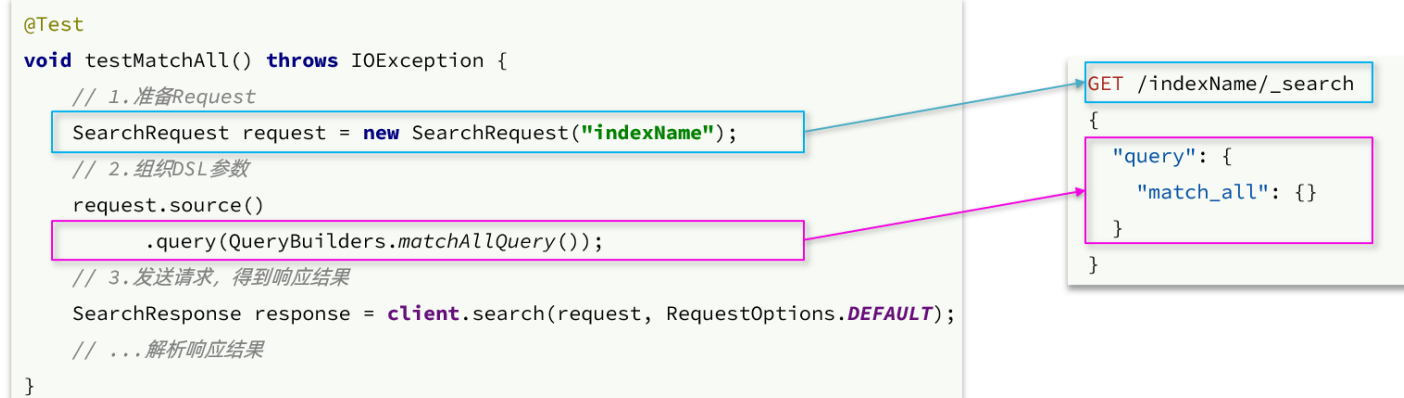
### 2.1.快速入门

之前说过，由于Elasticsearch对外暴露的接口都是Restful风格的接口，因此JavaAPI调用就是在发送Http请求。而我们核心要做的就是利用利用Java代码组织请求参数，解析响应结果。

这个参数的格式完全参考DSL查询语句的JSON结构，因此我们在学习的过程中，会不断的把JavaAPI与DSL语句对比。大家在学习记忆的过程中，也应该这样对比学习。

### 2.1.1.发送请求

首先以 `match_all` 查询为例，其DSL和JavaAPI的对比如图：



代码解读：

- 第一步，创建 `SearchRequest` 对象，指定索引库名
- 第二步，利用 `request.source()` 构建DSL，DSL中可以包含查询、分页、排序、高亮等
  - `query()`：代表查询条件，利用 `QueryBuilders.matchAllQuery()` 构建一个 `match_all` 查询的DSL
- 第三步，利用 `client.search()` 发送请求，得到响应

这里关键的API有两个，一个是 `request.source()`，它构建的就是DSL中的完整JSON参数。其中包含了 `query`、`sort`、`from`、`size`、`highlight` 等所有功能：

```
SearchRequest request = new SearchRequest( ...indices: "" );
```

```
request.source().
```

```
m aggregation(AggregationBuilder aggregation)
```

```
var
```

```
m sort(SortBuilder<?> sort)
```

```
m highlighter(HighlightBuilder highlightBuilder)
```

```
m sort(String name, SortOrder order)
```

```
m aggregation(PipelineAggregationBuilder aggregation)
```

```
m highlighter()
```

```
m sort(String name)
```

```
m sort(List<SortBuilder<?>> sorts)
```

```
m size(int size)
```

```
m size()
```

```
m from(int from)
```

```
m from()
```

```
m query(QueryBuilder query)
```

```
m query()
```

黑马程序员-研究院

另一个是 `QueryBuilders`，其中包含了我们学习过的各种叶子查询、复合查询等：

QueryBuilders

```
m matchAllQuery(): MatchAllQueryBuilder
```

```
m matchQuery(String, Object): MatchQueryBuilder
```

```
m idsQuery(): IdsQueryBuilder
```

```
m termQuery(String, String): TermQueryBuilder
```

```
m rangeQuery(String): RangeQueryBuilder
```

```
m wildcardQuery(String, String): WildcardQueryBuilder
```

```
m simpleQueryStringQuery(String): SimpleQueryStringBuilder
```

```
m boostingQuery(QueryBuilder, QueryBuilder): BoostingQueryBuilder
```

```
m boolQuery(): BoolQueryBuilder
```

```
m functionScoreQuery(QueryBuilder): FunctionScoreQueryBuilder
```

```
m geoDistanceQuery(String): GeoDistanceQueryBuilder
```

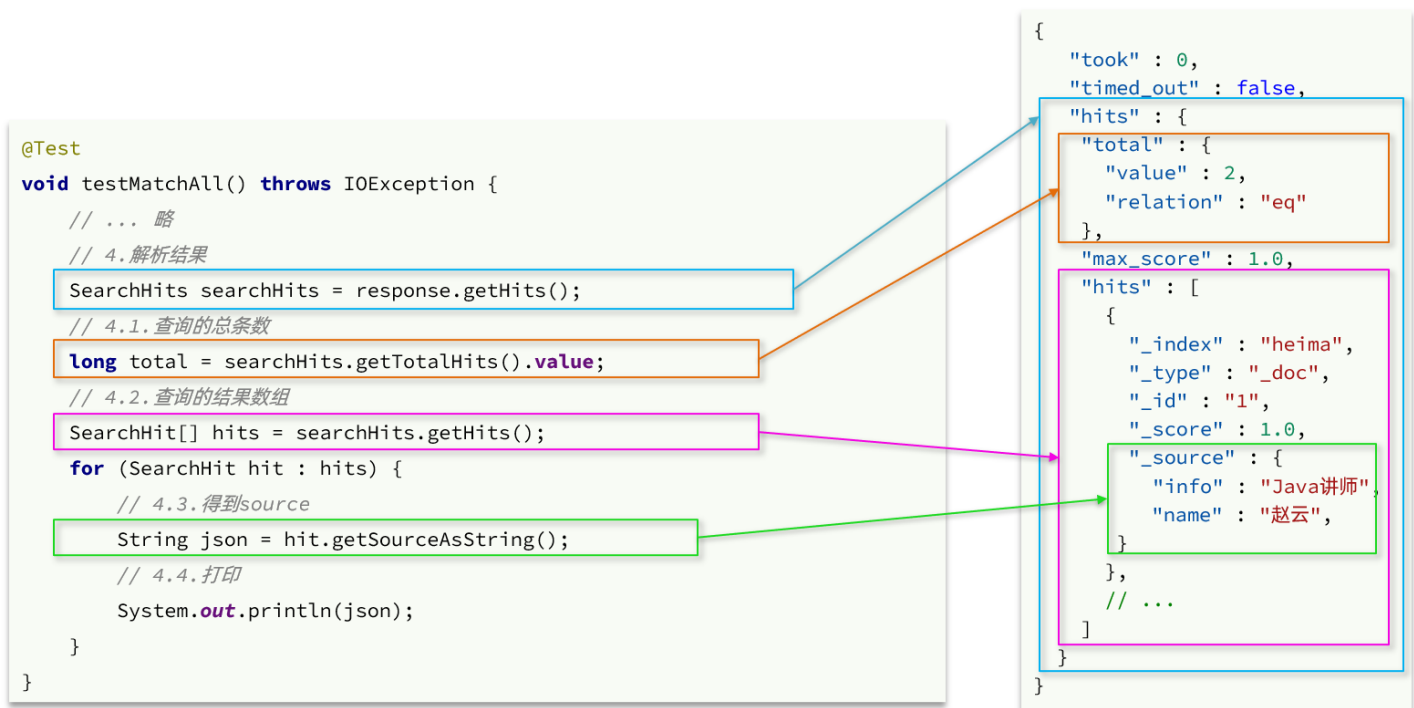
黑马程序员-研究院

## 2.1.2.解析响应结果

在发送请求以后，得到了响应结果 `SearchResponse`，这个类的结构与我们在kibana中看到的响应结果JSON结构完全一致：

```
1 {
2     "took" : 0,
3     "timed_out" : false,
4     "hits" : {
5         "total" : {
6             "value" : 2,
7             "relation" : "eq"
8         },
9         "max_score" : 1.0,
10        "hits" : [
11            {
12                "_index" : "heima",
13                "_type" : "_doc",
14                "_id" : "1",
15                "_score" : 1.0,
16                "_source" : {
17                    "info" : "Java讲师",
18                    "name" : "赵云"
19                }
20            }
21        ]
22    }
23 }
```

因此，我们解析 `SearchResponse` 的代码就是在解析这个JSON结果，对比如下：



## 代码解读：

elasticsearch返回的结果是一个JSON字符串，结构包含：

- `hits`：命中的结果
  - `total`：总条数，其中的value是具体的总条数值
  - `max_score`：所有结果中得分最高的文档的相关性算分
  - `hits`：搜索结果的文档数组，其中的每个文档都是一个json对象
    - `_source`：文档中的原始数据，也是json对象

因此，我们解析响应结果，就是逐层解析JSON字符串，流程如下：

- `SearchHits`：通过 `response.getHits()` 获取，就是JSON中的最外层的 `hits`，代表命中的结果
  - `SearchHits#getTotalHits().value`：获取总条数信息
  - `SearchHits#getHits()`：获取 `SearchHit` 数组，也就是文档数组
    - `SearchHit#getSourceAsString()`：获取文档结果中的 `_source`，也就是原始的 `json` 文档数据

## 2.1.3.总结

文档搜索的基本步骤是：



1. 创建 `SearchRequest` 对象
2. 准备 `request.source()`，也就是DSL。
  - a. `QueryBuilders` 来构建查询条件
  - b. 传入 `request.source()` 的 `query()` 方法
3. 发送请求，得到结果
4. 解析结果（参考JSON结果，从外到内，逐层解析）

完整代码如下：

```
1 @Test
2 void testMatchAll() throws IOException {
3     // 1.创建Request
4     SearchRequest request = new SearchRequest("items");
5     // 2.组织请求参数
6     request.source().query(QueryBuilders.matchAllQuery());
7     // 3.发送请求
8     SearchResponse response = client.search(request, RequestOptions.DEFAULT);
9     // 4.解析响应
10    handleResponse(response);
11 }
12
13 private void handleResponse(SearchResponse response) {
14     SearchHits searchHits = response.getHits();
15     // 1.获取总条数
16     long total = searchHits.getTotalHits().value;
17     System.out.println("共搜索到" + total + "条数据");
18     // 2.遍历结果数组
19     SearchHit[] hits = searchHits.getHits();
20     for (SearchHit hit : hits) {
21         // 3.得到_source, 也就是原始json文档
22         String source = hit.getSourceAsString();
23         // 4.反序列化并打印
24         ItemDoc item = JSONUtil.toBean(source, ItemDoc.class);
25         System.out.println(item);
26     }
27 }
```

## 2.2.叶子查询

所有的查询条件都是由QueryBuilders来构建的，叶子查询也不例外。因此整套代码中变化的部分仅仅是query条件构造的方式，其它不动。

例如 `match` 查询：

```
1 @Test
2 void testMatch() throws IOException {
3     // 1.创建Request
4     SearchRequest request = new SearchRequest("items");
5     // 2.组织请求参数
6     request.source().query(QueryBuilders.matchQuery("name", "脱脂牛奶"));
7     // 3.发送请求
8     SearchResponse response = client.search(request, RequestOptions.DEFAULT);
9     // 4.解析响应
10    handleResponse(response);
11 }
```

再比如 `multi_match` 查询：

```
1 @Test
2 void testMultiMatch() throws IOException {
3     // 1.创建Request
4     SearchRequest request = new SearchRequest("items");
5     // 2.组织请求参数
6     request.source().query(QueryBuilders.multiMatchQuery("脱脂牛奶", "name",
7     "category"));
8     // 3.发送请求
9     SearchResponse response = client.search(request, RequestOptions.DEFAULT);
10    // 4.解析响应
11    handleResponse(response);
12 }
```

还有 `range` 查询：

```
1 @Test
2 void testRange() throws IOException {
3     // 1.创建Request
4     SearchRequest request = new SearchRequest("items");
5     // 2.组织请求参数
6
7     request.source().query(QueryBuilders.rangeQuery("price").gte(10000).lte(30000))
8     ;
```

```

7      // 3.发送请求
8      SearchResponse response = client.search(request, RequestOptions.DEFAULT);
9      // 4.解析响应
10     handleResponse(response);
11 }

```

还有 `term` 查询：

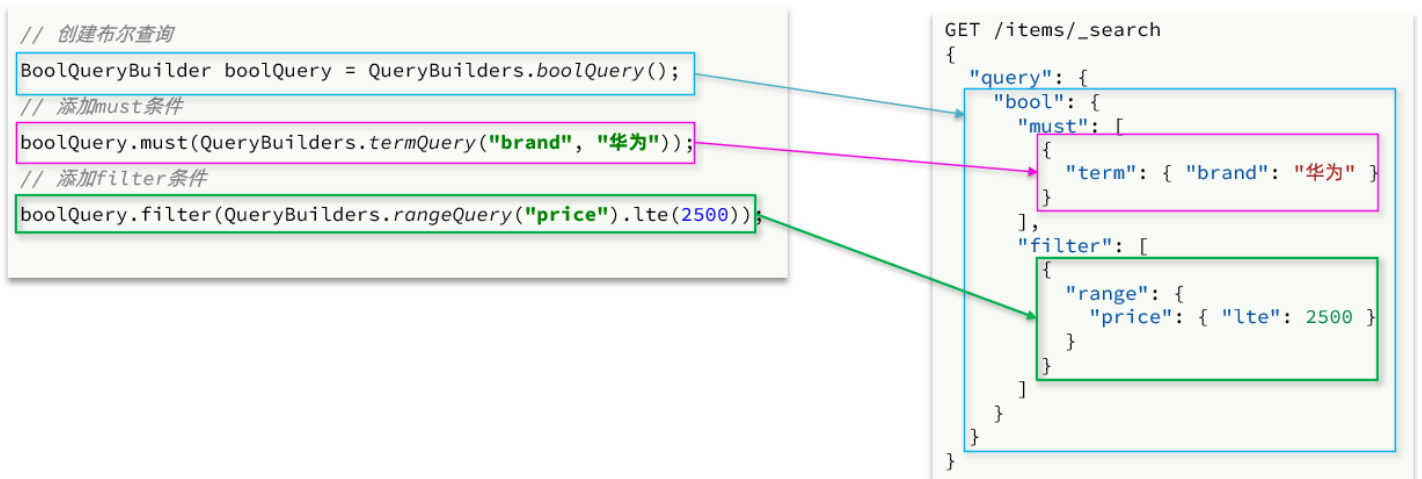
```

1  @Test
2  void testTerm() throws IOException {
3      // 1.创建Request
4      SearchRequest request = new SearchRequest("items");
5      // 2.组织请求参数
6      request.source().query(QueryBuilders.termQuery("brand", "华为"));
7      // 3.发送请求
8      SearchResponse response = client.search(request, RequestOptions.DEFAULT);
9      // 4.解析响应
10     handleResponse(response);
11 }

```

## 2.3.复合查询

复合查询也是由 `QueryBuilders` 来构建，我们以 `bool` 查询为例，DSL和JavaAPI的对比如图：



完整代码如下：

```

1  @Test
2  void testBool() throws IOException {
3      // 1.创建Request

```

```

4      SearchRequest request = new SearchRequest("items");
5      // 2.组织请求参数
6      // 2.1.准备bool查询
7      BoolQueryBuilder bool = QueryBuilders.boolQuery();
8      // 2.2.关键字搜索
9      bool.must(QueryBuilders.matchQuery("name", "脱脂牛奶"));
10     // 2.3.品牌过滤
11     bool.filter(QueryBuilders.termQuery("brand", "德亚"));
12     // 2.4.价格过滤
13     bool.filter(QueryBuilders.rangeQuery("price").lte(30000));
14     request.source().query(bool);
15     // 3.发送请求
16     SearchResponse response = client.search(request, RequestOptions.DEFAULT);
17     // 4.解析响应
18     handleResponse(response);
19 }

```

## 2.4.排序和分页

之前说过，`request.source()` 就是整个请求JSON参数，所以排序、分页都是基于这个来设置，其DSL和JavaAPI的对比如下：

```

// 查询
request.source().query(QueryBuilders.matchAllQuery());
// 分页
request.source().from(0).size(5);
// 价格排序
request.source().sort("price", SortOrder.ASC);

```

```

GET /indexName/_search
{
  "query": {
    "match_all": {}
  },
  "from": 0,
  "size": 5,
  "sort": [
    {
      "FIELD": "desc"
    }
  ]
}

```

完整示例代码：

```

1 @Test
2 void testPageAndSort() throws IOException {
3     int pageNo = 1, pageSize = 5;
4
5     // 1.创建Request
6     SearchRequest request = new SearchRequest("items");
7     // 2.组织请求参数
8     // 2.1.搜索条件参数
9     request.source().query(QueryBuilders.matchQuery("name", "脱脂牛奶"));

```

```

10 // 2.2.排序参数
11 request.source().sort("price", SortOrder.ASC);
12 // 2.3.分页参数
13 request.source().from((pageNo - 1) * pageSize).size(pageSize);
14 // 3.发送请求
15 SearchResponse response = client.search(request, RequestOptions.DEFAULT);
16 // 4.解析响应
17 handleResponse(response);
18 }

```

## 2.5.高亮

高亮查询与前面的查询有两点不同：

- 条件同样是在 `request.source()` 中指定，只不过高亮条件要基于 `HighlightBuilder` 来构造
- 高亮响应结果与搜索的文档结果不在一起，需要单独解析

首先来看高亮条件构造，其DSL和JavaAPI的对比如图：

```

request.source().highlighter(
    SearchSourceBuilder.highlight()
        .field("name")
        .preTags("<em>")
        .postTags("</em>")
);

```

```

GET /hotel/_search
{
  "query": {
    "match": {
      "name": "脱脂牛奶"
    }
  },
  "highlight": {
    "fields": {
      "name": {
        "pre_tags": "<em>",
        "post_tags": "</em>"
      }
    }
  }
}

```

示例代码如下：

```

1 @Test
2 void testHighlight() throws IOException {
3     // 1.创建Request
4     SearchRequest request = new SearchRequest("items");
5     // 2.组织请求参数

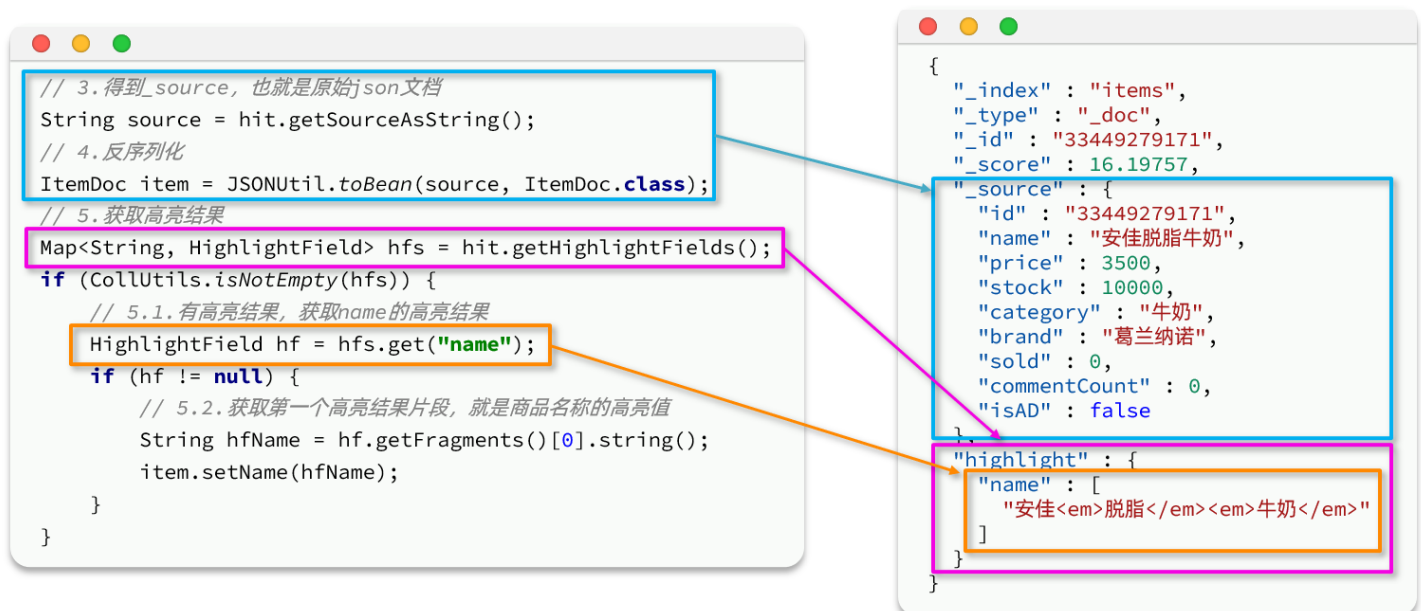
```

```

6      // 2.1.query条件
7      request.source().query(QueryBuilders.matchQuery("name", "脱脂牛奶"));
8      // 2.2.高亮条件
9      request.source().highlighter(
10         SearchSourceBuilder.highlight()
11             .field("name")
12             .preTags("<em>")
13             .postTags("</em>")
14     );
15     // 3.发送请求
16     SearchResponse response = client.search(request, RequestOptions.DEFAULT);
17     // 4.解析响应
18     handleResponse(response);
19 }

```

再来看结果解析，文档解析的部分不变，主要是高亮内容需要单独解析出来，其DSL和JavaAPI的对比如图：



### 代码解读：

- 第 3、4 步：从结果中获取 `_source`。 `hit.getSourceAsString()`，这部分是非高亮结果，json字符串。还需要反序列为 `ItemDoc` 对象
- 第 5 步：获取高亮结果。 `hit.getHighlightFields()`，返回值是一个 `Map`，key是高亮字段名称，值是 `HighlightField` 对象，代表高亮值
- 第 5.1 步：从 `Map` 中根据高亮字段名称，获取高亮字段值对象 `HighlightField`
- 第 5.2 步：从 `HighlightField` 中获取 `Fragments`，并且转为字符串。这部分就是真正的高亮字符串了

- 最后：用高亮的结果替换 `ItemDoc` 中的非高亮结果

完整代码如下：

```
1 private void handleResponse(SearchResponse response) {
2     SearchHits searchHits = response.getHits();
3     // 1.获取总条数
4     long total = searchHits.getTotalHits().value;
5     System.out.println("共搜索到" + total + "条数据");
6     // 2.遍历结果数组
7     SearchHit[] hits = searchHits.getHits();
8     for (SearchHit hit : hits) {
9         // 3.得到_source, 也就是原始json文档
10        String source = hit.getSourceAsString();
11        // 4.反序列化
12        ItemDoc item = JSONUtil.toBean(source, ItemDoc.class);
13        // 5.获取高亮结果
14        Map<String, HighlightField> hfs = hit.getHighlightFields();
15        if (CollUtils.isEmpty(hfs)) {
16            // 5.1.有高亮结果, 获取name的高亮结果
17            HighlightField hf = hfs.get("name");
18            if (hf != null) {
19                // 5.2.获取第一个高亮结果片段, 就是商品名称的高亮值
20                String hfName = hf.getFragments()[0].string();
21                item.setName(hfName);
22            }
23        }
24        System.out.println(item);
25    }
26 }
```

### 3.数据聚合

聚合（`aggregations`）可以让我们极其方便的实现对数据的统计、分析、运算。例如：

- 什么品牌的手机最受欢迎？
- 这些手机的平均价格、最高价格、最低价格？
- 这些手机每月的销售情况如何？

实现这些统计功能的比数据库的sql要方便的多，而且查询速度非常快，可以实现近实时搜索效果。

官方文档：

## Aggregations | Elasticsearch Guide [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Elasticsearch Guide [

聚合常见的有三类：

- **桶 ( Bucket )** 聚合：用来对文档做分组
  - `TermAggregation`：按照文档字段值分组，例如按照品牌值分组、按照国家分组
  - `Date Histogram`：按照日期阶梯分组，例如一周为一组，或者一月为一组
- **度量 ( Metric )** 聚合：用以计算一些值，比如：最大值、最小值、平均值等
  - `Avg`：求平均值
  - `Max`：求最大值
  - `Min`：求最小值
  - `Stats`：同时求 `max`、`min`、`avg`、`sum` 等
- **管道 ( pipeline )** 聚合：其它聚合的结果为基础做进一步运算

**注意：**参加聚合的字段必须是keyword、日期、数值、布尔类型

### 3.1.DSL实现聚合

与之前的搜索功能类似，我们依然先学习DSL的语法，再学习JavaAPI.

#### 3.1.1.Bucket聚合

例如我们要统计所有商品中共有哪些商品分类，其实就是以分类（category）字段对数据分组。category值一样的放在同一组，属于 `Bucket` 聚合中的 `Term` 聚合。

基本语法如下：

```
1 GET /items/_search
2 {
3   "size": 0,
4   "aggs": {
5     "category_agg": {
6       "terms": {
7         "field": "category",
8         "size": 20
```



```
9      }
10     }
11  }
12 }
```

语法说明：

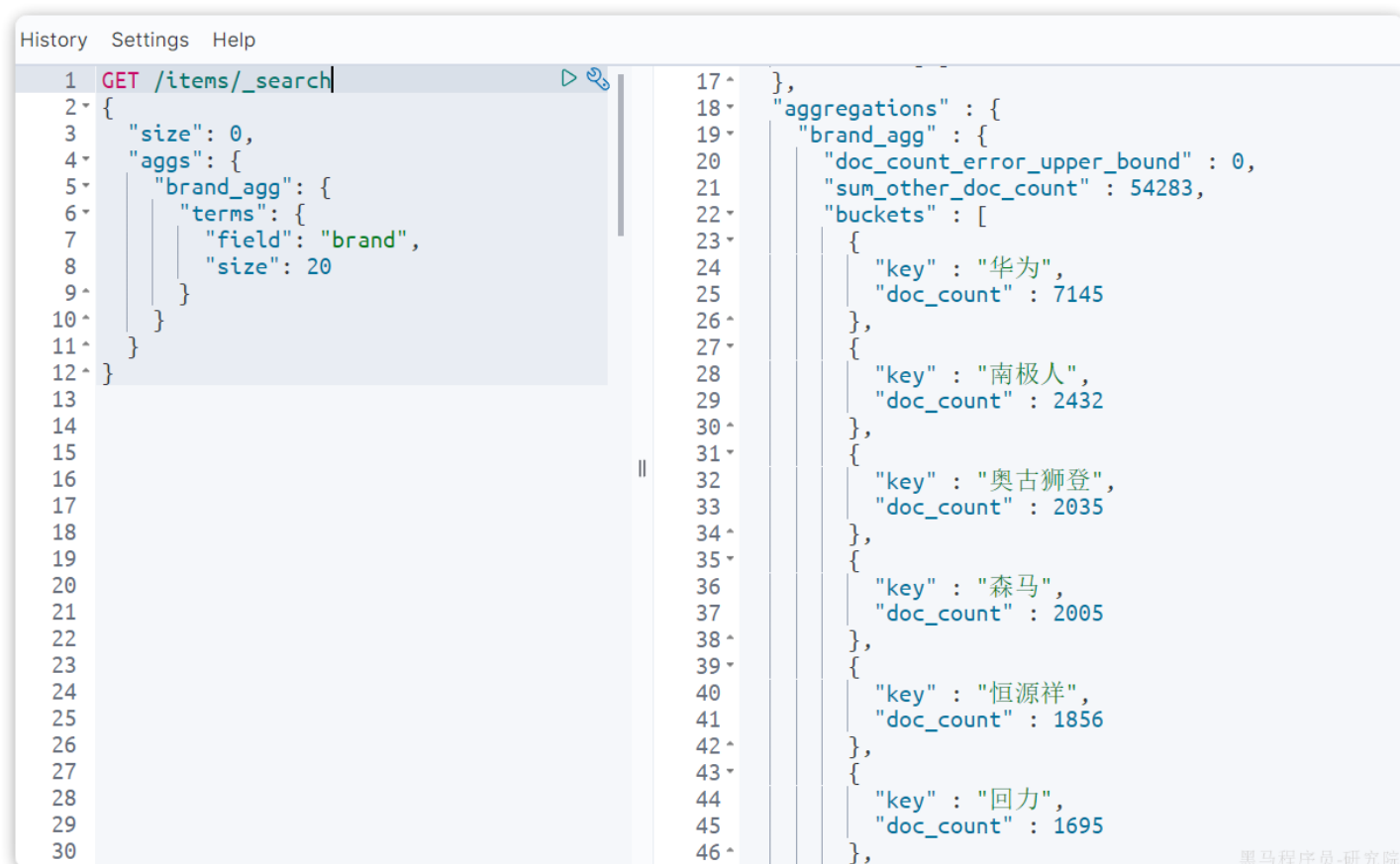
- `size`：设置 `size` 为0，就是每页查0条，则结果中就不包含文档，只包含聚合
- `aggs`：定义聚合
  - `category_agg`：聚合名称，自定义，但不能重复
    - `terms`：聚合的类型，按分类聚合，所以用 `term`
      - `field`：参与聚合的字段名称
      - `size`：希望返回的聚合结果的最大数量

来看下查询的结果：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : { },
  "hits" : {
    "total" : {
      "value" : 10000,
      "relation" : "gte"
    },
    "max_score" : null,
    "hits" : [ ]
  },
  "aggregations" : {
    "category_agg" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [ 聚合结果的bucket数组
        {
          "key" : "休闲鞋",      分类为“休闲鞋”的桶
          "doc_count" : 20612
        },
        {
          "key" : "牛仔裤",      分类为“牛仔裤”的桶
          "doc_count" : 19611
        },
        {
          "key" : "老花镜",      分类为“老花镜”的桶
          "doc_count" : 16222
        }
      ]
    }
  }
}
```

### 3.1.2.带条件聚合

默认情况下，Bucket聚合是对索引库的所有文档做聚合，例如我们统计商品中所有的品牌，结果如下：



```
1 GET /items/_search
2 {
3   "size": 0,
4   "aggs": {
5     "brand_agg": {
6       "terms": {
7         "field": "brand",
8         "size": 20
9       }
10    }
11  }
12 }

17 },
18 "aggregations": {
19   "brand_agg": {
20     "doc_count_error_upper_bound": 0,
21     "sum_other_doc_count": 54283,
22     "buckets": [
23       {
24         "key": "华为",
25         "doc_count": 7145
26       },
27       {
28         "key": "南极人",
29         "doc_count": 2432
30       },
31       {
32         "key": "奥古狮登",
33         "doc_count": 2035
34       },
35       {
36         "key": "森马",
37         "doc_count": 2005
38       },
39       {
40         "key": "恒源祥",
41         "doc_count": 1856
42       },
43       {
44         "key": "回力",
45         "doc_count": 1695
46       }
47     ]
48   }
49 }
```

可以看到统计出的品牌非常多。

但真实场景下，用户会输入搜索条件，因此聚合必须是对搜索结果聚合。那么聚合必须添加限定条件。

例如，我想知道价格高于3000元的手机品牌有哪些，该怎么统计呢？

我们需要从需求中分析出搜索查询的条件和聚合的目标：

- 搜索查询条件：
  - 价格高于3000
  - 必须是手机
- 聚合目标：统计的是品牌，肯定是对brand字段做term聚合

语法如下：

```
1 GET /items/_search
2 {
3   "query": {
4     "bool": {
```

```

5     "filter": [
6       {
7         "term": {
8           "category": "手机"
9         }
10      },
11      {
12        "range": {
13          "price": {
14            "gte": 300000
15          }
16        }
17      }
18    ]
19  },
20 },
21 "size": 0,
22 "aggs": {
23   "brand_agg": {
24     "terms": {
25       "field": "brand",
26       "size": 20
27     }
28   }
29 }
30 }

```

聚合结果如下：

```

1 {
2   "took" : 2,
3   "timed_out" : false,
4   "hits" : {
5     "total" : {
6       "value" : 13,
7       "relation" : "eq"
8     },
9     "max_score" : null,
10    "hits" : [ ]
11  },
12  "aggregations" : {
13    "brand_agg" : {
14      "doc_count_error_upper_bound" : 0,
15      "sum_other_doc_count" : 0,
16      "buckets" : [

```

```

17      {
18          "key" : "华为",
19          "doc_count" : 7
20      },
21      {
22          "key" : "Apple",
23          "doc_count" : 5
24      },
25      {
26          "key" : "小米",
27          "doc_count" : 1
28      }
29  ]
30  }
31  }
32  }
33

```

可以看到，结果中只剩下3个品牌了。

### 3.1.3.Metric聚合

上节课，我们统计了价格高于3000的手机品牌，形成了一个个桶。现在我们需要对桶内的商品做运算，获取每个品牌价格的最小值、最大值、平均值。

这就要用到 **Metric** 聚合了，例如 **stat** 聚合，就可以同时获取 **min**、**max**、**avg** 等结果。

语法如下：

```

1 GET /items/_search
2 {
3   "query": {
4     "bool": {
5       "filter": [
6         {
7           "term": {
8             "category": "手机"
9           }
10        },
11        {
12          "range": {
13            "price": {
14              "gte": 300000
15            }
16          }
17        }
18      ]
19    }
20  }
21 }

```

```

18     ]
19   }
20 },
21 "size": 0,
22 "aggs": {
23   "brand_agg": {
24     "terms": {
25       "field": "brand",
26       "size": 20
27     },
28     "aggs": {
29       "stats_meric": {
30         "stats": {
31           "field": "price"
32         }
33       }
34     }
35   }
36 }
37 }

```

`query` 部分就不说了，我们重点解读聚合部分语法。

可以看到我们在 `brand_agg` 聚合的内部，我们新加了一个 `aggs` 参数。这个聚合就是 `brand_agg` 的子聚合，会对 `brand_agg` 形成的每个桶中的文档分别统计。

- `stats_meric`：聚合名称
  - `stats`：聚合类型，stats是 `metric` 聚合的一种
    - `field`：聚合字段，这里选择 `price`，统计价格

由于stats是对brand\_agg形成的每个品牌桶内文档分别做统计，因此每个品牌都会统计出自己的价格最小、最大、平均值。

结果如下：

```

"aggregations" : {
  "brand_agg" : {
    "doc_count_error_upper_bound" : 0,
    "sum_other_doc_count" : 0,
    "buckets" : [
      {
        "key" : "华为",
        "doc_count" : 7,
        "stats_meric" : {
          "count" : 7,
          "min" : 349900.0,
          "max" : 549400.0,
          "avg" : 460585.71428571426,
          "sum" : 3224100.0
        }
      },
      {
        "key" : "Apple",
        "doc_count" : 5,
        "stats_meric" : {
          "count" : 5,
          "min" : 628900.0,
          "max" : 689900.0,
          "avg" : 653400.0,
          "sum" : 3267000.0
        }
      },
      {
        "key" : "小米",
        "doc_count" : 1,

```

品牌桶聚合

"华为"这个品牌的价格统计

"Apple"这个品牌的价格统计

黑马程序员-研究院

另外，我们还可以让聚合按照每个品牌的价格平均值排序：

```
History Settings Help
54 GET /items/_search
55 {
56   "query": { },
74   "size": 0,
75   "aggs": {
76     "brand_agg": {
77       "terms": {
78         "field": "brand",
79         "size": 20,
80         "order": {
81           "stats_meric.avg": "desc"
82         }
83       },
84       "aggs": {
85         "stats_meric": {
86           "stats": {
87             "field": "price"
88           }
89         }
90       }
91     }
92   }
93 }
94
18 "aggregations" : {
19   "brand_agg" : {
20     "doc_count_error_upper_bound" : 0,
21     "sum_other_doc_count" : 0,
22     "buckets" : [
23       {
24         "key" : "Apple",
25         "doc_count" : 5,
26         "stats_meric" : {
27           "count" : 5,
28           "min" : 628900.0,
29           "max" : 689900.0,
30           "avg" : 653400.0,
31           "sum" : 3267000.0
32         }
33       },
34       {
35         "key" : "小米",
36         "doc_count" : 1,
37         "stats_meric" : {
38           "count" : 1,
39           "min" : 584900.0,
40           "max" : 584900.0,
41           "avg" : 584900.0,
42           "sum" : 584900.0
43         }
44       },
45       {
46         "key" : "华为",
47         "doc_count" : 7,
```

### 3.1.4.总结

aggs代表聚合，与query同级，此时query的作用是？

- 限定聚合的文档范围

聚合必须的三要素：

- 聚合名称
- 聚合类型
- 聚合字段

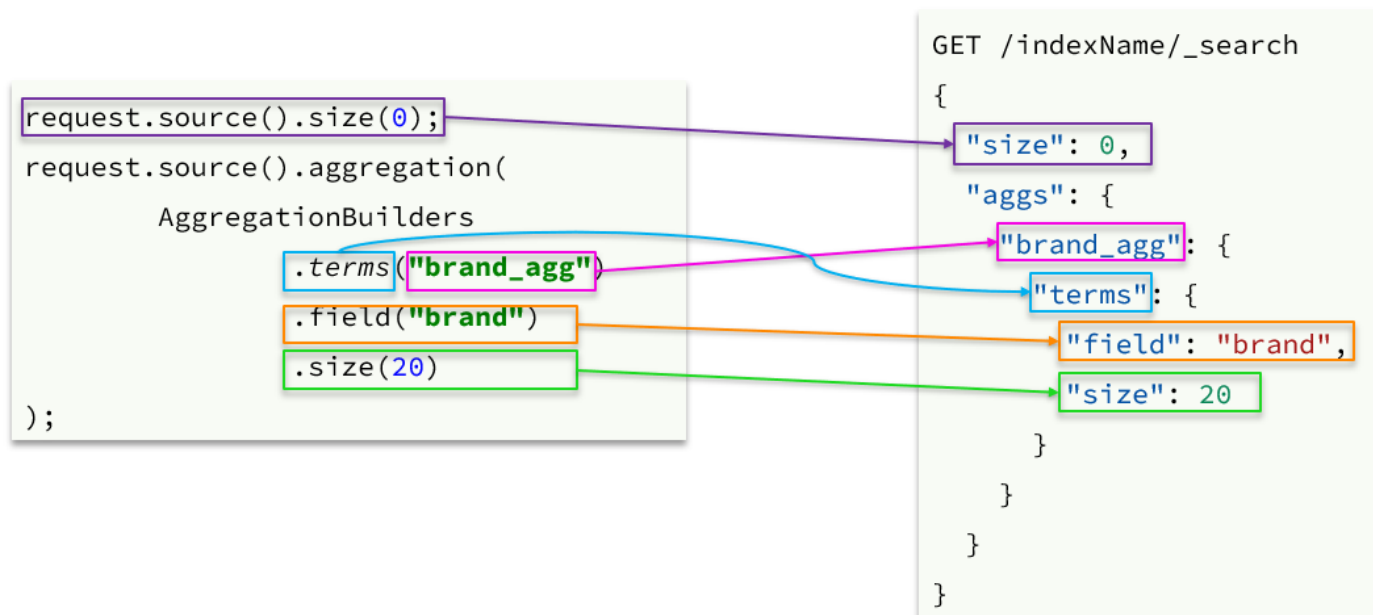
聚合可配置属性有：

- size：指定聚合结果数量
- order：指定聚合结果排序方式
- field：指定聚合字段

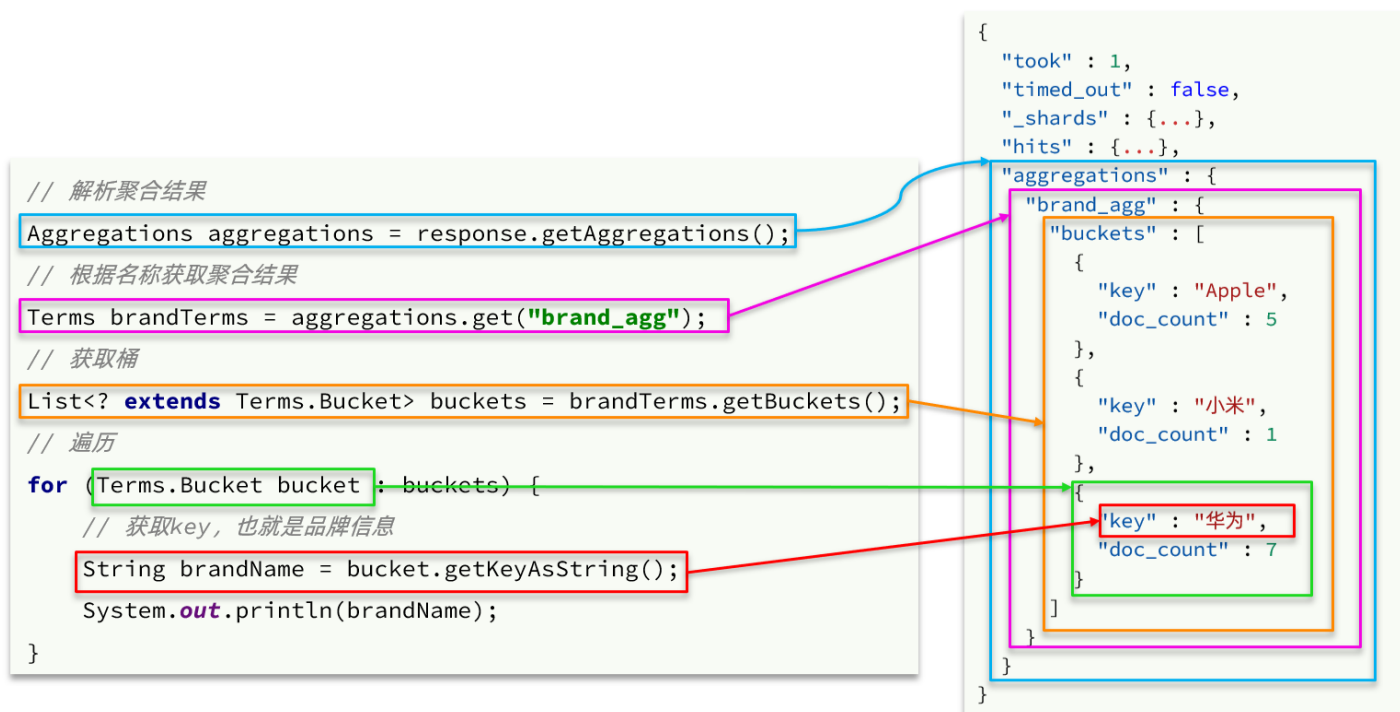
## 3.2.RestClient实现聚合

可以看到在DSL中，`aggs` 聚合条件与 `query` 条件是同一级别，都属于查询JSON参数。因此依然是利用 `request.source()` 方法来设置。

不过聚合条件的要利用 `AggregationBuilders` 这个工具类来构造。DSL与JavaAPI的语法对比如下：



聚合结果与搜索文档同一级别，因此需要单独获取和解析。具体解析语法如下：



完整代码如下：

```
1 @Test
2 void testAgg() throws IOException {
3     // 1.创建Request
4     SearchRequest request = new SearchRequest("items");
5     // 2.准备请求参数
```



```

6      BoolQueryBuilder bool = QueryBuilders.boolQuery()
7          .filter(QueryBuilders.termQuery("category", "手机"))
8          .filter(QueryBuilders.rangeQuery("price").gte(300000));
9      request.source().query(bool).size(0);
10     // 3.聚合参数
11     request.source().aggregation(
12         AggregationBuilders.terms("brand_agg").field("brand").size(5)
13     );
14     // 4.发送请求
15     SearchResponse response = client.search(request, RequestOptions.DEFAULT);
16     // 5.解析聚合结果
17     Aggregations aggregations = response.getAggregations();
18     // 5.1.获取品牌聚合
19     Terms brandTerms = aggregations.get("brand_agg");
20     // 5.2.获取聚合中的桶
21     List<? extends Terms.Bucket> buckets = brandTerms.getBuckets();
22     // 5.3.遍历桶内数据
23     for (Terms.Bucket bucket : buckets) {
24         // 5.4.获取桶内key
25         String brand = bucket.getKeyAsString();
26         System.out.print("brand = " + brand);
27         long count = bucket.getDocCount();
28         System.out.println("; count = " + count);
29     }
30 }

```

## 4.作业

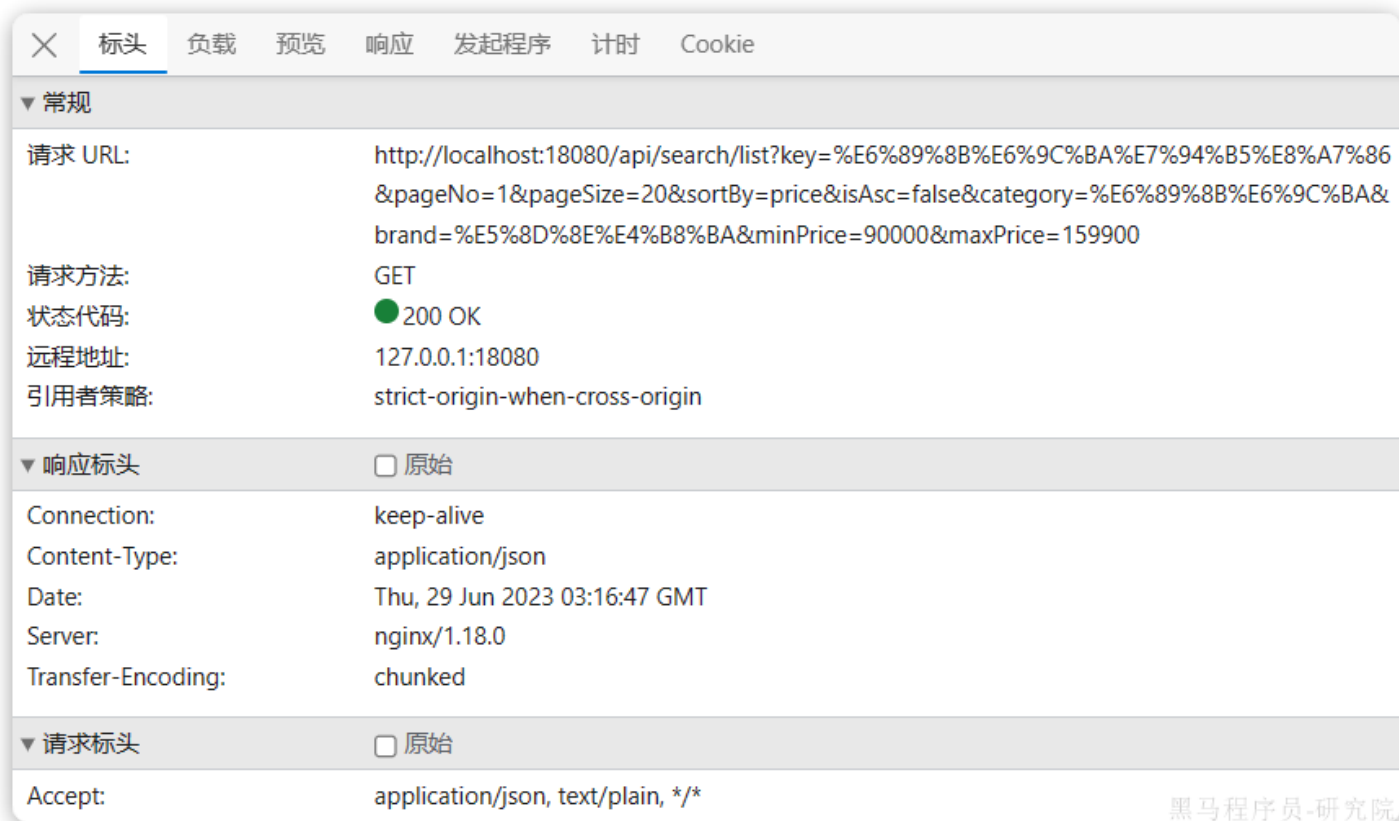
Elasticsearch的基本语法我们已经学完，足以应对大多数搜索业务需求了。接下来大家就可以基于学习的知识实现商品搜索的业务了。

在昨天的作业中要求大家拆分一个独立的微服务：`search-service`，在这个微服务中实现搜索数据的导入、商品数据库数据与elasticsearch索引库数据的同步。

接下来的搜索功能也要在 `search-service` 服务中实现。

### 4.1.实现搜索接口

在黑马商城的搜索页面，输入关键字，点击搜索时，会发现前端会发起查询商品的请求：



请求的接口信息如下：

- 请求方式：GET
- 请求路径：/search/list
- 请求参数：
  - key：搜索关键字
  - pageNo：页码
  - pageSize：每页大小
  - sortBy：排序字段
  - isAsc：是否升序
  - category：分类
  - brand：品牌
  - minPrice：价格最小值
  - maxPrice：价格最大值

请求参数可以参考原本 `item-service` 中

`com.hmall.item.controller.SearchController` 类中的基于数据库查询的接口：

```
hmall > item-service > src > main > java > com > hmall > item > controller > SearchController
SearchTest.testAgg
SearchController.java
18 @Api(tags = "搜索相关接口")
19 @RestController
20 @RequestMapping("/search")
21 @RequiredArgsConstructor
22 public class SearchController {
23
24     private final IItemService itemService;
25
26     @ApiOperation("搜索商品")
27     @GetMapping("/list")
28     public PageDTO<ItemDTO> search(ItemPageQuery query) {
29         // 分页查询
30         Page<Item> result = itemService.lambdaQuery()
31             .like(StrUtil.isNotBlank(query.getKey()), Item::getName, query.getKey())
32             .eq(StrUtil.isNotBlank(query.getBrand()), Item::getBrand, query.getBrand())
33             .eq(StrUtil.isNotBlank(query.getCategory()), Item::getCategory, query.getCategory())
34             .eq(Item::getStatus, val: 1)
35             .between(condition: query.getMaxPrice() != null, Item::getPrice, query.getMinPrice(),
36                 .page(query.toMpPage( defaultSortBy: "update_time", isAsc: false));
37         // 封装并返回
38         return PageDTO.of(result, ItemDTO.class);
39     }
40 }
```

## 4.2.过滤条件聚合

搜索页面的过滤项目是写死的：



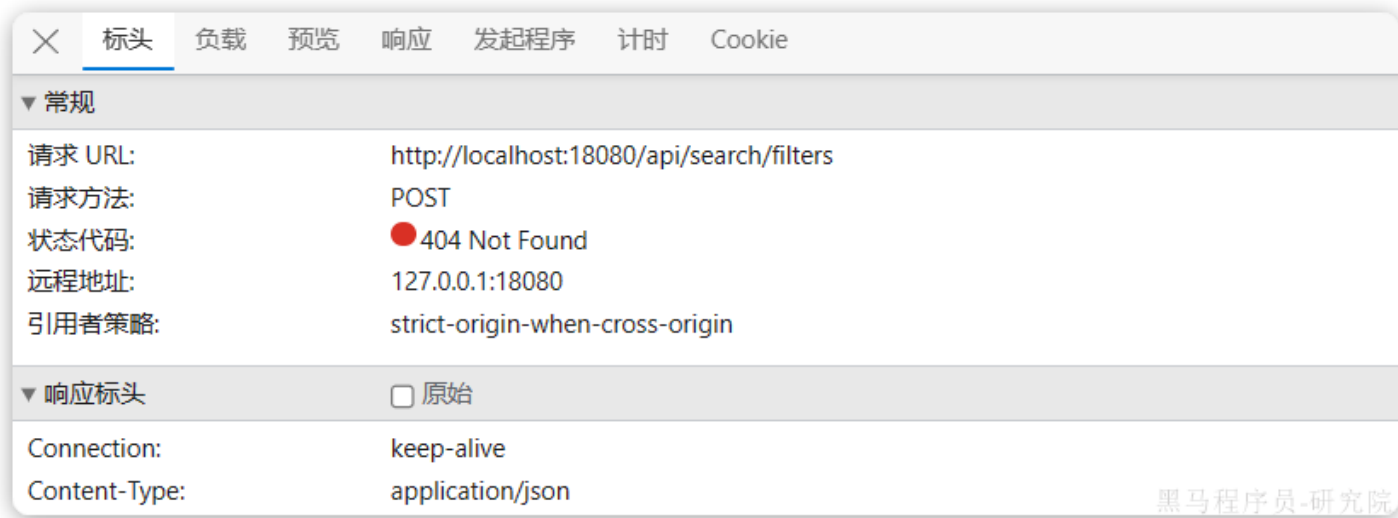
但是大家思考一下，随着搜索条件的变化，过滤条件展示的过滤项是不是应该跟着变化。

例如搜索 **电视**，那么搜索结果中展示的肯定只有电视，而此时过滤条件中的**分类**就不能还出现手机、拉杆箱等内容。过滤条件的**品牌**中就不能出现与电视无关的品牌。而是应该展示搜索结果中存在的分类和品牌。

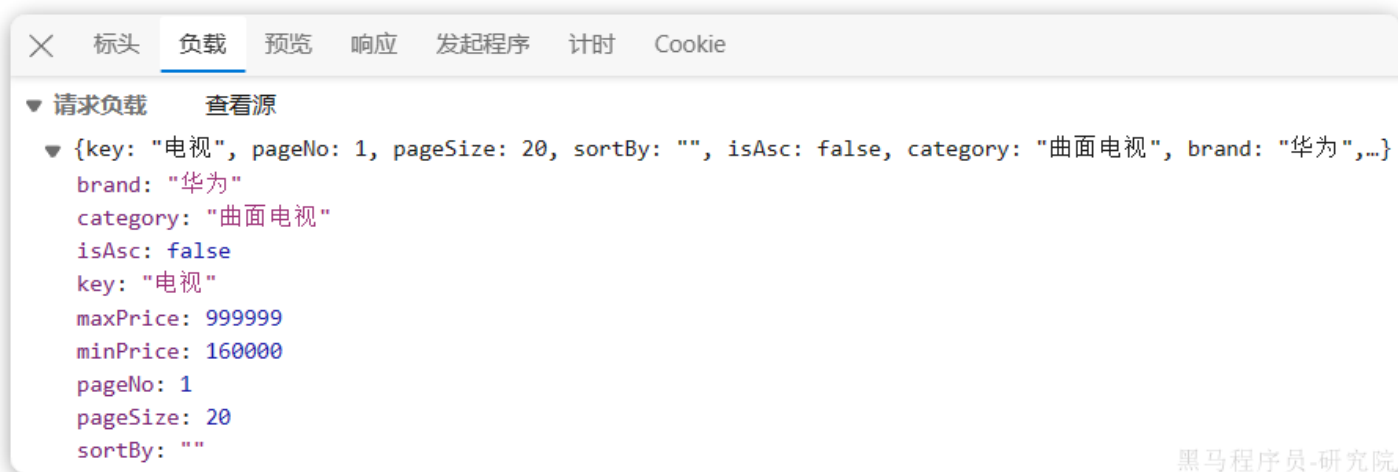
那么问题来，我们怎么知道搜索结果中存在哪些分类和品牌呢？

大家应该能想到，就是利用聚合，而且是带有限定条件的聚合。用户搜索的条件是什么，我们在对分类、品牌聚合时的条件也就是什么，这样就能统计出搜索结果中包含的分类、品牌了。

事实上，搜索时，前端已经发出了请求，尝试搜索栏中除价格以外的过滤项：



由于采用的是POST请求，所以参数在请求体中：



接口信息如下：

- 请求方式： POST
- 请求路径： /search/filters
- 请求参数：
  - key： 搜索关键字
  - pageNo： 页码
  - pageSize： 每页大小
  - sortBy： 排序字段
  - isAsc： 是否升序
  - category： 分类
  - brand： 品牌
  - minPrice： 价格最小值

- **maxPrice**: 价格最大值

可见参数与搜索参数一致，不过这里大家可以忽略分页和排序参数。

返回值参考这个格式：

```
1 {  
2   "category": ["手机", "曲面电视", "拉杆箱", "休闲鞋", "休闲鞋", "硬盘", "真皮包"],  
3   "brand": ["希捷", "小米", "华为", "oppo", "新秀丽", "Apple", "锤子"]  
4 }
```

## 4.3.竞价排名

elasticsearch的默认排序规则是按照相关性打分排序，而这个打分是可以通过API来控制的。详情可以参考复合查询中的算分函数查询（1.3.1小节）

对应的JavaAPI可以参考文档：

<https://www.elastic.co/guide/en/elasticsearch/client/java-api/7.12/java-compound-queries.html>

### Compound queries | Java API (deprecated) [7.12] | Elastic

IMPORTANT : No additional bug fixes or documentation updates will be released for this version. For the latest information, see the current release documentation . Elastic Docs › Java API (deprecated)

在商品的数据库表中，已经设计了 `isAD` 字段来标记广告商品，请利用 `function_score` 查询在原本搜索的结果基础上，让这些 `isAD` 字段值为 `true` 的商品排名到最前面。