ElasticSearch的基本原理与用法

一、简介

ElasticSearch和Solr都是基于Lucene的搜索引擎，不过ElasticSearch天生支持分布式，而Solr是4.0版本后的SolrCloud才是分布式版本，Solr的分布式支持需要ZooKeeper的支持。

这里有一个详细的ElasticSearch和Solr的对比：http://solr-vs-elasticsearch.com/

语法参考：
Elasticsearch Java API
Elasticsearch Query DSL

ElasticSearch安装部署：http://nero.life/2017/10/27/Elasticsearch%E7%AC%94%E8%AE%B0-%E4%B8%80-%E5%AE%89%E8%A3%85%E9%83%A8%E7%BD%B2/

elasticsearch Docker：http://nero.life/2017/10/27/Elasticsearch%E7%AC%94%E8%AE%B0-%E4%BA%8C-Docker/

ElasticSearch笔记：http://nero.life/2017/10/27/Elasticsearch%E7%AC%94%E8%AE%B0/

二、基本用法

集群（Cluster）： ES是一个分布式的搜索引擎，一般由多台物理机组成。这些物理机，通过配置一个相同的cluster name，互相发现，把自己组织成一个集群。

节点（Node）：同一个集群中的一个Elasticsearch主机。

Node类型：

1）data node: 存储index数据。Data nodes hold data and perform data related operations such as CRUD, search, and aggregations.

2）client node: 不存储index，处理转发客户端请求到Data Node。

3）master node: 不存储index，集群管理，如管理路由信息（routing infomation），判断node是否available，当有node出现或消失时重定位分片（shards），当有node failure时协调恢复。（所有的master node会选举出一个master leader node）

详情参考：https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html

主分片（Primary shard）：索引（下文介绍）的一个物理子集。同一个索引在物理上可以切多个分片，分布到不同的节点上。分片的实现是Lucene 中的索引。

注意：ES中一个索引的分片个数是建立索引时就要指定的，建立后不可再改变。所以开始建一个索引时，就要预计数据规模，将分片的个数分配在一个合理的范围。

副本分片（Replica shard）：每个主分片可以有一个或者多个副本，个数是用户自己配置的。ES会尽量将同一索引的不同分片分布到不同的节点上，提高容错性。对一个索引，只要不是所有shards所在的机器都挂了，就还能用。

索引（Index）：逻辑概念，一个可检索的文档对象的集合。类似与DB中的database概念。同一个集群中可建立多个索引。比如，生产环境常见的一种方法，对每个月产生的数据建索引，以保证单个索引的量级可控。

类型（Type）：索引的下一级概念，大概相当于数据库中的table。同一个索引里可以包含多个 Type。

文档（Document）：即搜索引擎中的文档概念，也是ES中一个可以被检索的基本单位，相当于数据库中的row，一条记录。

字段（Field）：相当于数据库中的column。ES中，每个文档，其实是以json形式存储的。而一个文档可以被视为多个字段的集合。比如一篇文章，可能包括了主题、摘要、正文、作者、时间等信息，每个信息都是一个字段，最后被整合成一个json串，落地到磁盘。

映射（Mapping）：相当于数据库中的schema，用来约束字段的类型，不过 Elasticsearch 的 mapping 可以不显示地指定、自动根据文档数据创建。

Database(数据库)	Index(索引)
Table(表)	Type(类型)
Row(行)	Document(文档)
Column(列)	Field(字段)
Schema(方案)	Mapping(映射)
Index(索引)	Everthing Indexed by default(所有字段都被索引)
SQL(结构化查询语言)	Query DSL(查询专用语言)

Elasticsearch集群可以包含多个索引（indices），每一个索引可以包含多个类型（types），每一个类型包含多个文档（documents），然后每个文档包含多个字段（Fields），这种面向文档型的储存，也算是NoSQL的一种吧。

ES比传统关系型数据库，对一些概念上的理解：

```
Relational DB -> Databases -> Tables -> Rows -> Columns
Elasticsearch -> Indices   -> Types  -> Documents -> Fields
```

从创建一个Client到添加、删除、查询等基本用法：

1、创建Client

```
1    public ElasticSearchService(String ipAddress, int port) {
2        client = new TransportClient()
3                .addTransportAddress(new InetSocketTransportAddress(ipAddress,
4                        port));
5    }
```

这里是一个TransportClient。

ES下两种客户端对比：

TransportClient：轻量级的Client，使用Netty线程池，Socket连接到ES集群。本身不加入到集群，只作为请求的处理。

Node Client：客户端节点本身也是ES节点，加入到集群，和其他ElasticSearch节点一样。频繁的开启和关闭这类Node Clients会在集群中产生“噪音”。

2、创建/删除Index和Type信息

```
1    // 创建索引
2    public void createIndex() {
3        client.admin().indices().create(new CreateIndexRequest(IndexName))
4                .actionGet();
5    }
6
7    // 清除所有索引
8    public void deleteIndex() {
9        IndicesExistsResponse indicesExistsResponse = client.admin().indices()
10               .exists(new IndicesExistsRequest(new String[] { IndexName }))
11               .actionGet();
12       if (indicesExistsResponse.isExists()) {
13           client.admin().indices().delete(new DeleteIndexRequest(IndexName))
14                   .actionGet();
15       }
16   }
17
18   // 删除Index下的某个Type
19   public void deleteType(){
20       client.prepareDelete().setIndex(IndexName).setType(TypeName).execute().actionGet();
21   }
```

```
22
23      // 定义索引的映射类型
24      public void defineIndexTypeMapping() {
25          try {
26              XContentBuilder mapBuilder = XContentFactory.jsonBuilder();
27              mapBuilder.startObject()
28                      .startObject(TypeName)
29                      .startObject("_all").field("enabled", false).endObject()
30                      .startObject("properties")
31                      .startObject(IDFieldName).field("type", "long").endObject()
32                      .startObject(SeqNumFieldName).field("type", "long").endObject()
33                      .startObject(IMSIFieldName).field("type", "string").field("index",
"not_analyzed").endObject()
34                      .startObject(IMEIFieldName).field("type", "string").field("index",
"not_analyzed").endObject()
35                      .startObject(DeviceIDFieldName).field("type", "string").field("index",
"not_analyzed").endObject()
36                      .startObject(OwnAreaFieldName).field("type", "string").field("index",
"not_analyzed").endObject()
37                      .startObject(TeleOperFieldName).field("type", "string").field("index",
"not_analyzed").endObject()
38                      .startObject(TimeFieldName).field("type", "date").field("store", "yes").endObject()
39                      .endObject()
40                      .endObject()
41                      .endObject();
42
43              PutMappingRequest putMappingRequest = Requests
44                      .putMappingRequest(IndexName).type(TypeName)
45                      .source(mapBuilder);
46              client.admin().indices().putMapping(putMappingRequest).actionGet();
47          } catch (IOException e) {
48              log.error(e.toString());
49          }
50      }
```

这里自定义了某个Type的索引映射（Mapping）：

1）默认ES会自动处理数据类型的映射：针对整型映射为long，浮点数为double，字符串映射为string，时间为date，true或false为boolean。

2）字段的默认配置是indexed，但不是stored的，也就是 field("index", "yes").field("store", "no")。

3）这里Disable了“_all”字段，_all字段会把所有的字段用空格连接，然后用“analyzed”的方式index这个字段，这个字段可以被search，但是不能被retrieve。

4）针对string，ES默认会做“analyzed”处理，即先做分词、去掉stop words等处理再index。如果你需要把一个字符串做为整体被索引到，需要把这个字段这样设置：field("index", "not_analyzed")。

5）默认_source字段是enabled，_source字段存储了原始Json字符串（original JSON document body that was passed at index time）。

详情参考：

https://www.elastic.co/guide/en/elasticsearch/guide/current/mapping-intro.html

https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-store.html

https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-all-field.html

https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-source-field.html

3、索引数据

```
1       // 批量索引数据
2       public void indexHotSpotDataList(List<Hotspotdata> dataList) {
3           if (dataList != null) {
4               int size = dataList.size();
5               if (size > 0) {
6                   BulkRequestBuilder bulkRequest = client.prepareBulk();
7                   for (int i = 0; i < size; ++i) {
8                       Hotspotdata data = dataList.get(i);
9                       String jsonSource = getIndexDataFromHotspotData(data);
10                      if (jsonSource != null) {
11                          bulkRequest.add(client
12                                  .prepareIndex(IndexName, TypeName,
13                                          data.getId().toString())
14                                  .setRefresh(true).setSource(jsonSource));
15                      }
16                  }
17
18                  BulkResponse bulkResponse = bulkRequest.execute().actionGet();
19                  if (bulkResponse.hasFailures()) {
20                      Iterator<BulkItemResponse> iter = bulkResponse.iterator();
21                      while (iter.hasNext()) {
22                          BulkItemResponse itemResponse = iter.next();
23                          if (itemResponse.isFailed()) {
24                              log.error(itemResponse.getFailureMessage());
25                          }
26                      }
27                  }
28              }
29          }
30      }
31
32      // 索引数据
33      public boolean indexHotspotData(Hotspotdata data) {
34          String jsonSource = getIndexDataFromHotspotData(data);
35          if (jsonSource != null) {
36              IndexRequestBuilder requestBuilder = client.prepareIndex(IndexName,
37                      TypeName).setRefresh(true);
38              requestBuilder.setSource(jsonSource)
39                      .execute().actionGet();
40              return true;
41          }
42
43          return false;
44      }
45
46      // 得到索引字符串
47      public String getIndexDataFromHotspotData(Hotspotdata data) {
48          String jsonString = null;
49          if (data != null) {
50              try {
51                  XContentBuilder jsonBuilder = XContentFactory.jsonBuilder();
52                  jsonBuilder.startObject().field(IDFieldName, data.getId())
53                          .field(SeqNumFieldName, data.getSeqNum())
54                          .field(IMSIFieldName, data.getImsi())
55                          .field(IMEIFieldName, data.getImei())
56                          .field(DeviceIDFieldName, data.getDeviceID())
57                          .field(OwnAreaFieldName, data.getOwnArea())
58                          .field(TeleOperFieldName, data.getTeleOper())
59                          .field(TimeFieldName, data.getCollectTime())
60                          .endObject();
61                  jsonString = jsonBuilder.string();
62              } catch (IOException e) {
63                  log.equals(e);
64              }
65          }
66
67          return jsonString;
68      }
```

ES支持批量和单个数据索引。

4、查询获取数据

```
1       // 获取少量数据100个
2       private List<Integer> getSearchData(QueryBuilder queryBuilder) {
3           List<Integer> ids = new ArrayList<>();
4           SearchResponse searchResponse = client.prepareSearch(IndexName)
5                   .setTypes(TypeName).setQuery(queryBuilder).setSize(100)
6                   .execute().actionGet();
7           SearchHits searchHits = searchResponse.getHits();
```

2019年11月(2)
2019年10月(4)
2019年9月(4)
2019年8月(4)
2019年7月(4)
2019年6月(7)
2019年5月(4)
2019年4月(22)
2019年3月(24)
更多

最新评论

1. Re:@Autowired用法详解

好

--mathum

2. Re:Java提高篇——对象克隆（复制）

数组也是引用类型嘛，假如类里面含有数组成员变量，不采用序列化的方式，如何实现深度克隆呢。

--Foring

3. Re:C++经典排序算法总结

@hahaha123456 不应该啊，快速排序是一种插旗子的方式，把旗子查到一个位置，然后分成左右两个区间再进行分治处理。左区间为[left,i-1],右区间为[j+1,right]...

--Boblim

4. Re:C++经典排序算法总结

快排中QuickSort(h,j+1,right); 第二个参数是不是应该是i+1?

--hahaha123456

5. Re:MySQL中授权(grant)和撤销授权(revoke

写的很棒通俗易懂

--安全客

阅读排行榜

1. C++中的STL中map用法详解(528037)
2. Linux常用命令大全(456800)
3. @Autowired用法详解(261946)
4. mybatis之foreach用法(227148)
5. JAVA中获取当前系统时间(217710)

评论排行榜

1. mybatis之foreach用法(8)
2. C++中的STL中map用法详解(7)
3. @Autowired用法详解(6)
4. C++经典排序算法总结(6)
5. C++ 获取文件夹下的所有文件名(6)

推荐排行榜

1. C++中的STL中map用法详解(54)
2. Linux常用命令大全(38)
3. C++中的inline用法(30)
4. mybatis之foreach用法(20)
5. C++宏定义详解(20)

```
        for (SearchHit searchHit : searchHits) {
            Integer id = (Integer) searchHit.getSource().get("id");
            ids.add(id);
        }
        return ids;
    }

    // 获取大量数据
    private List<Integer> getSearchDataByScrolls(QueryBuilder queryBuilder) {
        List<Integer> ids = new ArrayList<>();
        // 一次获取100000数据
        SearchResponse scrollResp = client.prepareSearch(IndexName)
                .setSearchType(SearchType.SCAN).setScroll(new TimeValue(60000))
                .setQuery(queryBuilder).setSize(100000).execute().actionGet();
        while (true) {
            for (SearchHit searchHit : scrollResp.getHits().getHits()) {
                Integer id = (Integer) searchHit.getSource().get(IDFieldName);
                ids.add(id);
            }
            scrollResp = client.prepareSearchScroll(scrollResp.getScrollId())
                    .setScroll(new TimeValue(600000)).execute().actionGet();
            if (scrollResp.getHits().getHits().length == 0) {
                break;
            }
        }

        return ids;
    }
```

这里的QueryBuilder是一个查询条件，ES支持分页查询获取数据，也可以一次性获取大量数据，需要使用Scroll Search。

5、聚合（Aggregation Facet）查询

```
    // 得到某段时间内设备列表上每个设备的数据分布情况<设备ID，数量>
    public Map<String, String> getDeviceDistributedInfo(String startTime,
                                                         String endTime, List<String> deviceList) {

        Map<String, String> resultsMap = new HashMap<>();

        QueryBuilder deviceQueryBuilder = getDeviceQueryBuilder(deviceList);
        QueryBuilder rangeBuilder = getDateRangeQueryBuilder(startTime, endTime);
        QueryBuilder queryBuilder = QueryBuilders.boolQuery()
                .must(deviceQueryBuilder).must(rangeBuilder);

        TermsBuilder termsBuilder = AggregationBuilders.terms("DeviceIDAgg").size(Integer.MAX_VALUE)
                .field(DeviceIDFieldName);
        SearchResponse searchResponse = client.prepareSearch(IndexName)
                .setQuery(queryBuilder).addAggregation(termsBuilder)
                .execute().actionGet();
        Terms terms = searchResponse.getAggregations().get("DeviceIDAgg");
        if (terms != null) {
            for (Terms.Bucket entry : terms.getBuckets()) {
                resultsMap.put(entry.getKey(),
                        String.valueOf(entry.getDocCount()));
            }
        }
        return resultsMap;
    }
```

Aggregation查询可以查询类似统计分析这样的功能：如某个月的数据分布情况，某类数据的最大、最小、总和、平均值等。

详情参考：https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/java-aggs.html

查询准备

当我们获取到connection以后，接下来就可以开始做查询的准备。

- 1 首先我们可以通过client来获取到一个SearchRequestBuilder的实例，这个是我们查询的主干。
  然后你需要给SearchRequestBuilder指定查询目标, Index以及Type（映射到数据库就是数据库名称以及表名）
- 2 指定分页信息，setFrom以及setSize。

PS: 千万不要指望能一次性的查询到所有的数据，尤其是document特别多的时候。Elasticsearch最多只会给你返回1000条数据，一次性返回过量的数据是灾难性，这点无论是Elasticsearch亦或是Database都适用。

```
SearchRequestBuilder searchRequestBuilder = client.prepareSearch(ES_ITEM_INDEX).setTypes(ES_ITEM_TYPE)
                .setFrom((pageNum - 1) * pageSize)
                .setSize(pageSize);
```

- 3 接下来，你可能需要指定返回的字段，可以用如下的代码设置：

```
String[] includes = {"id", "name"};
searchRequestBuilder.setFetchSource(includes, null);
```

- 4 构建查询器

我们以自己熟悉的SQL来做实例讲解，如下sql:

```
select * from student where id = '123' and age > 12 and name like '小明' and hid in (...)
```

我们看SQL后面where条件。包含了’等于，大于，模糊查询，in查询’。

首先我们需要一个能包含复杂查询条件的BoolQueryBuilder，你可以将一个个小查询条件设置进去，最后将它设给searchRequestBuilder。

```
BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
```

id = ‘123’：

```
QueryBuilder idQueryBuilder = QueryBuilders.termQuery("id", 123);
boolQueryBuilder.must(idQueryBuilder);
```

age > 12:

```
RangeQueryBuilder ageQueryBuilder = QueryBuilders.rangeQuery("age").gt(12);
boolQueryBuilder.must(ageQueryBuilder);
```

name like ‘小明’：

```
MatchQueryBuilder matchQueryBuilder = QueryBuilders.matchQuery("name", '小明');
matchQueryBuilder.operator(Operator.AND);
boolQueryBuilder.must(matchQueryBuilder);
```

这个就是我们比较关注的关键字查询了。因为我的Elasticsearch中字段name，已经配置了ik_smart分词器。
所以此处会将我的条件”小明”进行ik_smart的分词查询。而我设置的Operator.AND属性，指的是必须要我传入的查询条件分词后字段全部匹配，才会返回结果。默认是Operator.OR，也就是其中有一个匹配，结果就返回。

hid in (...):

```
QueryBuilder queryBuilder = QueryBuilders.termsQuery("hid", hidList);
boolQueryBuilder.must(queryBuilder);
```

PS: 上述的SQL条件中，皆是以AND关键字进行拼接。如果是其他的，比如OR，比如Not in呢？请看下面：

> and – must
> or – should
> not in – must not

好，最后我们将queryBuilder设置进searchRequestBuilder中。

```
// 设置查询条件
searchRequestBuilder.setQuery(boolQueryBuilder);
```

- 5 排序
  我们同样需要一个排序构造器：SortBuilder。

咱先根据age字段，做降序。如下代码：

```
SortBuilder sortBuilder = SortBuilders.fieldSort("age");
sortBuilder.order(SortOrder.fromString("DESC"));
```

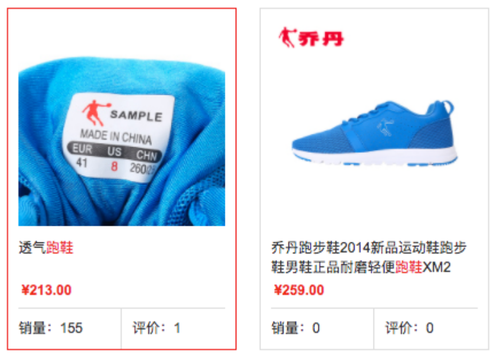
如果想要做一些较为复杂的排序，比如多个字段相加：

```
String scriptStr = "doc['clickCount'].value + doc['soldNum'].value";
sortBuilder = SortBuilders.scriptSort(new Script(scriptStr), ScriptSortBuilder.ScriptSortType.NUMBER);
```

```
// 设置排序规则
searchRequestBuilder.addSort(sortBuilder);
```

- 6 高亮显示

在具体业务查询中，其实我们通常需要用到如该图的效果。



将查询命中的关键，进行标记颜色样式。

Elasticsearch同样提供了功能的实现，我们只需要设置被查询的字段即可。如下代码：

```
HighlightBuilder highlightBuilder = new HighlightBuilder();
highlightBuilder.field("name");
searchRequestBuilder.highlighter(highlightBuilder);
```

在下面的**解析结果**中，会具体介绍怎么解析出这么高亮结果。

- 7 聚合

当然，Elasticsearch也提供了强大的聚合功能：

```
// 年龄聚合计算
TermsAggregationBuilder ageTermsAggregationBuilder = AggregationBuilders.terms("ageAgg");
brandTermsAggregationBuilder.field("age");
brandTermsAggregationBuilder.size(1000);
searchRequestBuilder.addAggregation(ageTermsAggregationBuilder);
```

- 8 执行查询

```
// 开始执行查询
SearchResponse searchResponse = searchRequestBuilder.execute().actionGet();
```

- 9 解析结果

直接上代码：

```
SearchHits searchHits = searchResponse.getHits();
List<Student> students = new ArrayList<>();
for (SearchHit hit : searchHits.getHits()) {
    String json = hit.getSourceAsString();
    # 读取json，转为成对象
    Student student= ...
    // 关键字高亮显示
    HighlightField highlightField = hit.getHighlightFields().get("name");
    String highName = student.getName();
    if (highlightField != null) {
        highName = highlightField.getFragments()[0].toString();
    }
    student.setHighName(highName);
    students.add(student);
}
```

我们可以从searchResponse中，获取到所有的命中目标列表。然后循环列表，每个命中的hit,可以直接转化成json。
然后可以json转化工具，映射到自己的bean上。
其中，当你设置了highlightBuilder以后，你可以在每个hit里，get到被标签包裹着的String。
（ps: 默认是em，你也可以设置其他的标签，以供前端渲染。）

使用QueryBuilders进行查询

elastcisearch-query-builder接受配置文件(特定json格式)或者json格式的字符串配置，配置格式如下：

```
{
  "index": "user_portrait",
  "type": "docs",
  "from": "${from}",
  "size": "10",
  "include_source": ["name","age"],  //需要哪些字段
  "exclude_source": ["sex"],  //排除哪些字段
  "query_type": "terms_level_query",
  "terms_level_query": {
    "terms_level_type": "term_query",
    "term_query": {
      "value": "${value}",
      "key": "key",
      "boost": 2
    }
  },
  "aggregations": [
    {
      "aggregation_type": "terms",
      "name": "",
      "field": "field",
      "sub_aggregations": {
        "aggregation_type": "terms",
        "name": "sub",
        "field": "field",
        "size": "${size.value}",
        "sort": "asc",
        "sort_by": "_count"
      }
    }
  ],
  "highlight":{
      "fields": [
            {
              "field": "content",
              "number_of_fragment": 2,
              "no_match_size": 150
            }
       ],
      "pre_tags":["<em>"],
      "post_tags":["</em>"]
  },
  "sort": [
    "_score",
    {
      "field": "age",
      "order": "asc"
    }
  ]
}
```

参数说明

 # index

index表示elasticSearch中的索引或者别名。

 # type

type表示elasticSearch索引或者别名下的type。

 # from

from表示检索文档时的偏移量，相当于关系型数据库里的offset。

 # include_source

include_source 搜索结果中包含某些字段，格式为json数组，"include_source": ["name","age"]。

 # exclude_source

exclude_source 搜索结果中排除某些字段，格式为json数组，"exclude_source":["sex"]。

 # query_type

query_type表示查询类型，支持三种类型terms_level_query,text_level_query,bool_level_query,并且这三种类型不可以一起使用。

- terms_level_query 操作的精确字段是存储在反转索引中的。这些查询通常用于结构化数据,
  如数字、日期和枚举, 而不是全文字段,包含term_query,terms_query,range_query,exists_query 等类型。
- text_level_query 查询通常用于在完整文本字段 (如电子邮件正文) 上运行全文查询。他们了解如何分析被查询的字段,并在执行之前将每个字段的分析器 (或 search_analyzer) 应用到查询字符串。

  包含 match_query,multi_match_query,query_string,simple_query_string 等类型。
- bool_query 与其他查询的布尔组合匹配的文档匹配的查询。bool 查询映射到 Lucene BooleanQuery。它是使用一个或多个布尔子句生成的, 每个子句都有一个类型化的实例。
  布尔查询的查询值包括: must,filter,should,must_not. 在每个布尔查询的查询类型值中,可以包含terms_level_query 和text_level_query中任意的查询类型，如此便可以构造非常复杂的查询情况。

QueryBuilder 是es中提供的一个查询接口, 可以对其进行参数设置来进行查询:

```
package com.wenbronk.javaes;

import java.net.InetSocketAddress;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map.Entry;

import org.elasticsearch.action.ListenableActionFuture;
import org.elasticsearch.action.get.GetRequestBuilder;
import org.elasticsearch.action.get.GetResponse;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.search.SearchType;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.text.Text;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
import org.elasticsearch.common.unit.TimeValue;
import org.elasticsearch.index.query.IndicesQueryBuilder;
import org.elasticsearch.index.query.NestedQueryBuilder;
import org.elasticsearch.index.query.QueryBuilder;
import org.elasticsearch.index.query.QueryBuilders;
import org.elasticsearch.index.query.QueryStringQueryBuilder;
import org.elasticsearch.index.query.RangeQueryBuilder;
import org.elasticsearch.index.query.SpanFirstQueryBuilder;
import org.elasticsearch.index.query.WildcardQueryBuilder;
import org.elasticsearch.search.SearchHit;
import org.elasticsearch.search.SearchHits;
import org.junit.Before;
import org.junit.Test;

/**
 * java操作查询api
 * @author 231
 *
 */
public class JavaESQuery {

    private TransportClient client;

    @Before
    public void testBefore() {
        Settings settings = Settings.settingsBuilder().put("cluster.name", "wenbronk_escluster").build();
        client = TransportClient.builder().settings(settings).build()
                .addTransportAddress(new InetSocketTransportAddress(new InetSocketAddress("192.168.50.37",
9300)));
        System.out.println("success to connect escluster");
    }

    /**
     * 使用get查询
     */
    @Test
    public void testGet() {
        GetRequestBuilder requestBuilder = client.prepareGet("twitter", "tweet", "1");
        GetResponse response = requestBuilder.execute().actionGet();
        GetResponse getResponse = requestBuilder.get();
        ListenableActionFuture<GetResponse> execute = requestBuilder.execute();
        System.out.println(response.getSourceAsString());
    }

    /**
     * 使用QueryBuilder
     * termQuery("key", obj) 完全匹配
     * termsQuery("key", obj1, obj2..)   一次匹配多个值
     * matchQuery("key", Obj) 单个匹配, field不支持通配符, 前缀具高级特性
     * multiMatchQuery("text", "field1", "field2"..);  匹配多个字段, field有通配符忒行
     * matchAllQuery();         匹配所有文件
     */
    @Test
    public void testQueryBuilder() {
//        QueryBuilder queryBuilder = QueryBuilders.termQuery("user", "kimchy");
        QueryBUilder queryBuilder = QueryBuilders.termQuery("user", "kimchy", "wenbronk", "vini");
        QueryBuilders.termsQuery("user", new ArrayList<String>().add("kimchy"));
//        QueryBuilder queryBuilder = QueryBuilders.matchQuery("user", "kimchy");
//        QueryBuilder queryBuilder = QueryBuilders.multiMatchQuery("kimchy", "user", "message", "gender");
        QueryBuilder queryBuilder = QueryBuilders.matchAllQuery();
        searchFunction(queryBuilder);

    }

    /**
     * 组合查询
     * must(QueryBuilders) :   AND
     * mustNot(QueryBuilders): NOT
     * should:                  : OR
     */
    @Test
    public void testQueryBuilder2() {
        QueryBuilder queryBuilder = QueryBuilders.boolQuery()
            .must(QueryBuilders.termQuery("user", "kimchy"))
            .mustNot(QueryBuilders.termQuery("message", "nihao"))
            .should(QueryBuilders.termQuery("gender", "male"));
        searchFunction(queryBuilder);
    }

    /**
     * 只查询一个id的
     * QueryBuilders.idsQuery(String...type).ids(Collection<String> ids)
     */
    @Test
    public void testIdsQuery() {
        QueryBuilder queryBuilder = QueryBuilders.idsQuery().ids("1");
        searchFunction(queryBuilder);
    }

    /**
     * 包裹查询, 高于设定分数, 不计算相关性
     */
    @Test
    public void testConstantScoreQuery() {
        QueryBuilder queryBuilder = QueryBuilders.constantScoreQuery(QueryBuilders.termQuery("name",
"kimchy")).boost(2.0f);
        searchFunction(queryBuilder);
        // 过滤查询
//        QueryBuilders.constantScoreQuery(FilterBuilders.termQuery("name", "kimchy")).boost(2.0f);

    }

    /**
     * disMax查询
     * 对子查询的结果做union, score沿用子查询score的最大值,
     * 广泛用于muti-field查询
     */
    @Test
```

```
    public void testDisMaxQuery() {
        QueryBuilder queryBuilder = QueryBuilders.disMaxQuery()
            .add(QueryBuilders.termQuery("user", "kimch"))  // 查询条件
            .add(QueryBuilders.termQuery("message", "hello"))
            .boost(1.3f)
            .tieBreaker(0.7f);
        searchFunction(queryBuilder);
    }

    /**
     * 模糊查询
     * 不能用通配符，不知道干啥用
     */
    @Test
    public void testFuzzyQuery() {
        QueryBuilder queryBuilder = QueryBuilders.fuzzyQuery("user", "kimch");
        searchFunction(queryBuilder);
    }

    /**
     * 父或子的文档查询
     */
    @Test
    public void testChildQuery() {
        QueryBuilder queryBuilder = QueryBuilders.hasChildQuery("sonDoc", QueryBuilders.termQuery("name",
"vini"));
        searchFunction(queryBuilder);
    }

    /**
     * moreLikeThisQuery: 实现基于内容推荐，支持实现一句话相似文章查询
     * {
        "more_like_this" : {
        "fields" : ["title", "content"],   // 要匹配的字段，不填默认_all
        "like_text" : "text like this one",   // 匹配的文本
        }

    percent_terms_to_match: 匹配项（term）的百分比，默认是0.3

    min_term_freq: 一篇文档中一个词语至少出现次数，小于这个值的词将被忽略，默认是2

    max_query_terms: 一条查询语句中允许最多查询词语的个数，默认是25

    stop_words: 设置停止词，匹配时会忽略停止词

    min_doc_freq: 一个词语最少在多少篇文档中出现，小于这个值的词会将被忽略，默认是无限制

    max_doc_freq: 一个词语最多在多少篇文档中出现，大于这个值的词会将被忽略，默认是无限制

    min_word_len: 最小的词语长度，默认是0

    max_word_len: 最多的词语长度，默认无限制

    boost_terms: 设置词语权重，默认是1

    boost: 设置查询权重，默认是1

    analyzer: 设置使用的分词器，默认是使用该字段指定的分词器
     */
    @Test
    public void testMoreLikeThisQuery() {
        QueryBuilder queryBuilder = QueryBuilders.moreLikeThisQuery("user")
                                .like("kimchy");
//                                .minTermFreq(1)          //最少出现的次数
//                                .maxQueryTerms(12);        // 最多允许查询的词语
        searchFunction(queryBuilder);
    }

    /**
     * 前缀查询
     */
    @Test
    public void testPrefixQuery() {
        QueryBuilder queryBuilder = QueryBuilders.matchQuery("user", "kimchy");
        searchFunction(queryBuilder);
    }

    /**
     * 查询解析查询字符串
     */
    @Test
    public void testQueryString() {
        QueryBuilder queryBuilder = QueryBuilders.queryStringQuery("+kimchy");
        searchFunction(queryBuilder);
    }

    /**
     * 范围内查询
     */
    public void testRangeQuery() {
        QueryBuilder queryBuilder = QueryBuilders.rangeQuery("user")
            .from("kimchy")
            .to("wenbronk")
            .includeLower(true)     // 包含上界
            .includeUpper(true);      // 包含下届
        searchFunction(queryBuilder);
    }

    /**
     * 跨度查询
     */
    @Test
    public void testSpanQueries() {
         QueryBuilder queryBuilder1 = QueryBuilders.spanFirstQuery(QueryBuilders.spanTermQuery("name", "葫芦
580娃"), 30000);     // Max查询范围的结束位置

         QueryBuilder queryBuilder2 = QueryBuilders.spanNearQuery()
                .clause(QueryBuilders.spanTermQuery("name", "葫芦580娃")) // Span Term Queries
                .clause(QueryBuilders.spanTermQuery("name", "葫芦3812娃"))
                .clause(QueryBuilders.spanTermQuery("name", "葫芦7139娃"))
                .slop(30000)                                    // Slop factor
                .inOrder(false)
                .collectPayloads(false);

        // Span Not
         QueryBuilder queryBuilder3 = QueryBuilders.spanNotQuery()
                .include(QueryBuilders.spanTermQuery("name", "葫芦580娃"))
                .exclude(QueryBuilders.spanTermQuery("home", "山西省太原市2552街道"));

        // Span Or
         QueryBuilder queryBuilder4 = QueryBuilders.spanOrQuery()
                .clause(QueryBuilders.spanTermQuery("name", "葫芦580娃"))
                .clause(QueryBuilders.spanTermQuery("name", "葫芦3812娃"))
                .clause(QueryBuilders.spanTermQuery("name", "葫芦7139娃"));

        // Span Term
         QueryBuilder queryBuilder5 = QueryBuilders.spanTermQuery("name", "葫芦580娃");
    }

    /**
     * 测试子查询
     */
    @Test
    public void testTopChildrenQuery() {
        QueryBuilders.hasChildQuery("tweet",
                QueryBuilders.termQuery("user", "kimchy"))
            .scoreMode("max");
    }

    /**
     * 通配符查询，支持 *
     * 匹配任何字符序列，包括空
     * 避免* 开始，会检索大量内容造成效率缓慢；对于通配符查询必须注意一个问题，就是参数必须小写，即例子中“kihy”必须小写，这是
个坑
     */
    @Test
```

```
    public void testWildCardQuery() {
        QueryBuilder queryBuilder = QueryBuilders.wildcardQuery("user", "ki*hy");
        searchFunction(queryBuilder);
    }

    /**
     * 嵌套查询，内嵌文档查询
     */
    @Test
    public void testNestedQuery() {
        QueryBuilder queryBuilder = QueryBuilders.nestedQuery("location",
                QueryBuilders.boolQuery()
                    .must(QueryBuilders.matchQuery("location.lat", 0.962590433140581))
                    .must(QueryBuilders.rangeQuery("location.lon").lt(36.0000).gt(0.000)))
        .scoreMode("total");

    }

    /**
     * 测试索引查询
     */
    @Test
    public void testIndicesQueryBuilder () {
        QueryBuilder queryBuilder = QueryBuilders.indicesQuery(
                QueryBuilders.termQuery("user", "kimchy"), "index1", "index2")
                .noMatchQuery(QueryBuilders.termQuery("user", "kimchy"));

    }



    /**
     * 查询遍历抽取
     * @param queryBuilder
     */
    private void searchFunction(QueryBuilder queryBuilder) {
        SearchResponse response = client.prepareSearch("twitter")
                .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
                .setScroll(new TimeValue(60000))
                .setQuery(queryBuilder)
                .setSize(100).execute().actionGet();

        while(true) {
            response = client.prepareSearchScroll(response.getScrollId())
                .setScroll(new TimeValue(60000)).execute().actionGet();
            for (SearchHit hit : response.getHits()) {
                Iterator<Entry<String, Object>> iterator = hit.getSource().entrySet().iterator();
                while(iterator.hasNext()) {
                    Entry<String, Object> next = iterator.next();
                    System.out.println(next.getKey() + ": " + next.getValue());
                    if(response.getHits().hits().length == 0) {
                        break;
                    }
                }
            }
            break;
        }
//        testResponse(response);
    }

    /**
     * 对response结果的分析
     * @param response
     */
    public void testResponse(SearchResponse response) {
        // 命中的记录数
        long totalHits = response.getHits().totalHits();

        for (SearchHit searchHit : response.getHits()) {
            // 打分
            float score = searchHit.getScore();
            // 文章id
            int id = Integer.parseInt(searchHit.getSource().get("id").toString());
            // title
            String title = searchHit.getSource().get("title").toString();
            // 内容
            String content = searchHit.getSource().get("content").toString();
            // 文章更新时间
            long updatetime = Long.parseLong(searchHit.getSource().get("updatetime").toString());
        }
    }

    /**
     * 对结果设置高亮显示
     */
    public void testHighLighted() {
        /*  5.0 版本后的高亮设置
         * client.#().#().highlighter(hBuilder).execute().actionGet();
        HighlightBuilder hBuilder = new HighlightBuilder();
        hBuilder.preTags("<h2>");
        hBuilder.postTags("</h2>");
        hBuilder.field("user");        // 设置高亮显示的字段
        */
        // 加入查询中
        SearchResponse response = client.prepareSearch("blog")
            .setQuery(QueryBuilders.matchAllQuery())
            .addHighlightedField("user")        // 添加高亮的字段
            .setHighlighterPreTags("<h1>")
            .setHighlighterPostTags("</h1>")
            .execute().actionGet();

        // 遍历结果，获取高亮片段
        SearchHits searchHits = response.getHits();
        for(SearchHit hit:searchHits){
            System.out.println("String方式打印文档搜索内容:");
            System.out.println(hit.getSourceAsString());
            System.out.println("Map方式打印高亮内容");
            System.out.println(hit.getHighlightFields());

            System.out.println("遍历高亮集合，打印高亮片段:");
            Text[] text = hit.getHighlightFields().get("title").getFragments();
            for (Text str : text) {
                System.out.println(str.string());
            }
        }
    }
}
```

elasticsearch查询api：match query:

match query

- 与**term query**不同，**match query**的查询词是被分词处理的（analyzed），即首先分词，然后构造相应的查询，所以应该确保查询的分词器和索引的分词器是一致的；
- 与**terms query**相似，提供的查询词之间默认是or的关系，可以通过operator属性指定；
- **match query**有两种形式，一种是简单形式，一种是bool形式；

match query VS match_phrase query

注意其差别：

- match query：会对查询语句进行分词，分词后查询语句中的任何一个词项被匹配，文档就会被搜索到。如果想查询匹配所有关键词的文档，可以用and操作符连接；
- match_phrase query：满足下面两个条件才会被搜索到
  - （1）分词后所有词项都要出现在该字段中
  - （2）字段中的词项顺序要一致

分类： java

标签: java

Boblim
关注 - 0
粉丝 - 451
+加关注

« 上一篇： Spring Boot中使用Swagger2自动构建API文档
» 下一篇： 使用Java High Level REST Client操作elasticsearch

posted @ 2019-04-22 12:50 Boblim 阅读(7267) 评论(0) 编辑 收藏 举报