【ELK 技术栈第二天 SpringDataElasticsearch】

主要内容

1. Spring Data ElasticSearch

学习目标

知识点	要求
Spring Data ElasticSearch	掌握

ELK 技术栈第二天 SpringDataElasticsearch

一、 Spring Data ElasticSearch

使用 Spring Data 下二级子项目 Spring Data Elasticsearch 进行操作。支持 POJO 方法操作 Elasticsearch。相比 Elasticsearch 提供的 API 更加简单更加方便。

1 修改 POM 文件添加依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
```

2 修改配置文件

集群版多地址之间使用逗号分隔。

在 ElasticSearch5.x 以前的版本中，客户端使用的是 Transport 客户端，通过 TCP 协议和 9300 端口访问 ES。在 6.x 及之后的版本中，官方推荐使用 Rest 客户端，通过 Http 协议和 9200 端口访问 ES。且在新版的 Spring Data Elasticsearch 框架中，Transport 客户端配置已经设置为过时配置，推荐使用 Rest 客户端。

2.1 高版本新客户端

```
spring.elasticsearch.rest.uris=http://192.168.40.141:9200
```

2.2 低版本常用客户端

```
spring.data.elasticsearch.cluster-name=elasticsearch
spring.data.elasticsearch.cluster-nodes= 192.168.40.141:9300
```

3 创建实体

@Document 指定实体类和索引对应关系

indexName：索引名称

type: 索引类型

shards: 主分片数量

replicas：复制分片数量

@Id 指定主键

@Field 指定普通属性

type： 对应 Elasticsearch 中属性类型。使用 FiledType 枚举可以快速获取。测试发现没有 type 属性可能出现无法自动创建类型问题，所以一定要有 type 属性。

text 类型能被分词

keywords 不能被分词

index： 是否创建索引。作为搜索条件时 index 必须为 true

analyzer：指定分词器类型。

fielddata：指定是否为 text 类型字段创建正向索引。默认为 false，设置为 true 则可以使用此字段排序。

```
@Document(indexName = "people_index",type = "people_type",shards =
1,replicas = 1)
public class People {
```

```
    @Id
    private String id;
    // 整个name 不被分词，切不创建索引
    // Keyword 表示不被分词
    @Field(type= FieldType.Keyword,index = false)
    private String name;
    // address 被ik 分词
    // Text 类型的属性才能被分词
    @Field(type = FieldType.Text,analyzer = "ik_max_word")
    private String address;

    @Field(type = FieldType.Long,index = false)
    private int age;
```

4 初始化索引

createIndex(): 创建索引，创建出来的索引是不带有 mapping 信息的。返回值表示是否创建成功

putMapping():为已有的索引添加 mapping 信息。不具备创建索引的能力。返回值表示是否创建成功

```
@SpringBootTest
class EsdemoApplicationTests {
    @Autowired
    private ElasticsearchTemplate elasticsearchTemplate;
    @Test
    void contextLoads() {
        boolean result1 = elasticsearchTemplate.createIndex(People.class);
        System.out.println(result1);
        boolean results = elasticsearchTemplate.putMapping(People.class);
        System.out.println(results);
    }

    @Autowired
    private ElasticsearchRestTemplate restTemplate;
    @Test
    public void testInitIndex(){
        // 创建索引，根据类型上的Document 注解创建
        boolean isCreated = restTemplate.createIndex(Item.class);
```

```
        // 设置映射，根据属性上的Field 注解设置
        boolean isMapped = restTemplate.putMapping(Item.class);
        System.out.println("创建索引是否成功：" + isCreated);
        System.out.println("设置映射是否成功：" + isMapped);
    }

}
```

5 删除索引

```
@Test
void delete(){
    boolean result = elasticsearchTemplate.deleteIndex(People.class);
    System.out.println(result);
}

/**
 * 删除索引
 */
@Test
public void deleteIndex(){
    // 扫描 Item 类型上的 Document 注解，删除对应的索引。
    boolean isDeleted = restTemplate.deleteIndex(Item.class);
    System.out.println("删除 Item 对应索引是否成功：" + isDeleted);
    // 直接删除对应名称的索引。
    isDeleted = restTemplate.deleteIndex("default_index");
    System.out.println("删除 default_index 索引是否成功：" + isDeleted);
}
```

6 添加文档

如果索引和类型不存在，也可以执行进行新增，新增后自动创建索引和类型。但是 field 通过动态 mapping 进行映射，elaticsearch 根据值类型进行判断每个属性类型，默认每个属性都是 standard 分词器，ik 分词器是不生效的。**所以一定要先通过代码进行初始化或直接在 elasticsearch 中通过命令创建所有 field 的 mapping**

6.1 新增单个文档

如果对象的 id 属性没有赋值，让 ES 自动生成主键，存储时 id 属性没有值，_id 存储 document 的主键值。

如果对象的 id 属性明确设置值，存储时 id 属性为设置的值，ES 中 document 对象的 _id 也是设置的值。

```
@Test
void insert1(){
    People peo = new People();
    peo.setId("123");

    peo.setName("张三");

    peo.setAddress("北京市海淀区回龙观东大街");

    peo.setAge(18);
    IndexQuery query = new IndexQuery();
    query.setObject(peo);
    String result = elasticsearchTemplate.index(query);
    System.out.println(result);
}

    @Test
    public void testInsert(){
        Item item = new Item();
        item.setId("111222333");
        item.setTitle("Spring In Action VI");
        item.setSellPoint("Spring 系列书籍，非常好的一本 Spring 框架学习手册。唯一缺点没有中文版。");
        item.setPrice(9900L);
        item.setNum(999);

        IndexQuery indexQuery =
                new IndexQueryBuilder() // 创建一个 IndexQuery 的构建器
                        .withObject(item) // 设置要新增的 Java 对象
                        .build(); // 构建 IndexQuery 类型的对象。

//        IndexQuery query = new IndexQuery();
//        query.setObject(item);
        // index 逻辑，相当于使用 PUT 请求，实现数据的新增。
        String result = restTemplate.index(indexQuery);
```

```
        System.out.println(result);
    }
```

6.2 批量新增

下面代码中使用的 IndexQueryBuilder()进行构建，可以一行代码完成。也可以使用上面的 IndexQuery()。效果是完全相同的，只是需要写多行。

```
@Test
void bulk(){
    List<IndexQuery> list = new ArrayList<>();
    // IndexQuery 多行写法
    IndexQuery indexQuery = new IndexQuery();
    indexQuery.setObject(new People("1","王五","北京东城",12));
    list.add(indexQuery);

    // IndexQuery 连缀写法
    list.add(new IndexQueryBuilder().withObject(new People("2","赵六","北京西城",13)).build());
    list.add(new IndexQueryBuilder().withObject(new People("3","吴七","北京昌平",14)).build());

    elasticsearchTemplate.bulkIndex(list);
}
/**
 * 批量新增
 * bulk 操作
 */
@Test
public void testBatchInsert(){
    List<IndexQuery> queries = new ArrayList<>();
    for(int i = 0; i < 3; i++){
        Item item = new Item();
        item.setId("2020"+i);
        item.setTitle("测试新增商品"+i);
        item.setSellPoint("测试新增商品卖点内容"+i);
        item.setPrice(new Random().nextLong());
        item.setNum(999);
        queries.add(
                new IndexQueryBuilder().withObject(item).build()
        );
```

```
    }
    // 批量新增，使用的是 bulk 操作。
    restTemplate.bulkIndex(queries);
}
```

7 删除操作

根据主键删除

delete(String indexName,String typeName,String id); 通过字符串指定索引，类型和 id 值

delete(Class,String id) 第一个参数传递实体类类类型，建议使用此方法，减少索引名和类型名由于手动编写出现错误的概率。

返回值为 delete 方法第二个参数值（删除文档的主键值）

```
@Test
void deleteDoc(){
    String result = elasticsearchTemplate.delete(People.class, "4");
    System.out.println(result);
}

    /**
     * 删除数据
     */
    @Test
    public void testDelete(){
        // 根据主键删除
        String result = restTemplate.delete(Item.class,
"epYDynEBHt6IU1hpAxvL");
        System.out.println(result);

        // 根据查询结果，删除查到的数据。 应用较少。
        /*DeleteQuery query = new DeleteQuery();
//        query.setIndex("ego_item");
//        query.setType("item");
        query.setQuery(
```

```
                QueryBuilders.matchQuery("title", "Spring In Action VI")
        );
        restTemplate.delete(query, Item.class);*/
    }
```

8 修改操作

修改操作就是新增代码，只要保证主键 id 已经存在，新增就是修改。

如果使用部分更新，则需要通过 update 方法实现。具体如下：

```
/**
 * 修改数据
 * 如果是全量替换，可以使用 index 方法实现，只要主键在索引中存在，就是全量替换。
 * 如果是部分修改，则可以使用 update 实现。
 */
@Test
public void testUpdate() throws Exception{
    UpdateRequest request = new UpdateRequest();
    request.doc(
            XContentFactory.jsonBuilder()
                    .startObject()
                    .field("name", "测试 update 更新数据，商品名称")
                    .endObject()
    );
    UpdateQuery updateQuery =
            new UpdateQueryBuilder()
                    .withUpdateRequest(request)
                    .withClass(Item.class)
                    .withId("20200")
                    .build();
    restTemplate.update(updateQuery);
}
```

9 搜索操作

9.1 模糊搜索

去所有 field 中搜索指定条件。

```
@Test
void query(){
    // NativeSearchQuery 构造方法参数。
    // 北京去和所有 field 进行匹配，只要出现了北京就可以进行查询
    QueryStringQueryBuilder queryStringQueryBuilder =
QueryBuilders.queryStringQuery("北京");
    // 查询条件 SearchQuery 是接口，只能实例化实现类。
    SearchQuery searchQuery = new
NativeSearchQuery(queryStringQueryBuilder);
    List<People> list = elasticsearchTemplate.queryForList(searchQuery,
People.class);
    for(People people : list){
        System.out.println(people);
    }
}
```

9.2 使用 match_all 搜索所有文档

```
@Test
void matchAll(){
    SearchQuery searchQuery = new
NativeSearchQuery(QueryBuilders.matchAllQuery());
    List<People> list = elasticsearchTemplate.queryForList(searchQuery,
People.class);
    for(People people : list){
        System.out.println(people);
    }
}
```

9.3 使用 match 搜索文档

```
@Test
void match(){
    SearchQuery searchQuery = new
NativeSearchQuery(QueryBuilders.matchQuery("address","我要去北京"));
    List<People> list = elasticsearchTemplate.queryForList(searchQuery,
People.class);
    for(People people : list){
        System.out.println(people);
    }
}
```

9.4 使用 match_phrase 搜索文档

短语搜索是对条件不分词，但是文档中属性根据配置实体类时指定的分词类型进行分词。

如果属性使用 ik 分词器，从分词后的索引数据中进行匹配。

```
@Test
void mathPhrase(){
    SearchQuery searchQuery = new
NativeSearchQuery(QueryBuilders.matchPhraseQuery("address","北京市"));
    List<People> list = elasticsearchTemplate.queryForList(searchQuery,
People.class);
    for(People people : list){
        System.out.println(people);
    }
}
```

9.5 使用 range 搜索文档

```
@Test
void range(){
    SearchQuery searchQuery = new
NativeSearchQuery(QueryBuilders.rangeQuery("age").gte(22).lte(23));
    List<People> list = elasticsearchTemplate.queryForList(searchQuery,
People.class);
    for(People people : list){
        System.out.println(people);
    }
}
```

9.6 多条件搜索

```
    @Test
    void MustShould(){
        BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
        List<QueryBuilder> listQuery = new ArrayList<>();

        listQuery.add(QueryBuilders.matchPhraseQuery("name","张三"));

        listQuery.add(QueryBuilders.rangeQuery("age").gte(22).lte(23));
//        boolQueryBuilder.should().addAll(listQuery); // 逻辑或
        boolQueryBuilder.must().addAll(listQuery);  // 逻辑与
        SearchQuery searchQuery = new NativeSearchQuery(boolQueryBuilder);
        List<People> list =
elasticsearchTemplate.queryForList(searchQuery, People.class);
        for(People people : list){
            System.out.println(people);
        }
    }
```

9.7 分页与排序

```
@Test
void PageSort(){
    SearchQuery searchQuery = new
NativeSearchQuery(QueryBuilders.matchAllQuery());
    // 分页 第一个参数是页码，从 0 算起。第二个参数是每页显示的条数
    searchQuery.setPageable(PageRequest.of(0,2));
```

```
    // 排序 第一个参数排序规则 DESC ASC 第二个参数是排序属性
    searchQuery.addSort(Sort.by(Sort.Direction.DESC,"age"));
    List<People> list = elasticsearchTemplate.queryForList(searchQuery,
People.class);
    for(People people : list){
        System.out.println(people);
    }
}
```

如果实体类中主键只有@Id 注解，String id 对应 ES 中是 text 类型，text 类型是不允许被排序，所以如果必须按照主键进行排序时需要在实体类中设置主键类型

```
@Id
@Field(type = FieldType.Keyword)
private String id;
```

9.8 高亮搜索

```
    @Test
    void hl() {
        // 高亮属性
        HighlightBuilder.Field hf = new HighlightBuilder.Field("address");
        // 高亮前缀
        hf.preTags("<span>");
        // 高亮后缀
        hf.postTags("</span>");

        NativeSearchQuery searchQuery = new NativeSearchQueryBuilder()
                // 排序
                .withSort(SortBuilders.fieldSort("age").order(SortOrder.D
ESC))
                // 分页
                .withPageable(PageRequest.of(0, 2))
                // 应用高亮
                .withHighlightFields(hf)
                // 查询条件， 必须要有条件，否则高亮

                .withQuery(QueryBuilders.matchQuery("address", "北京市
")).build();

        AggregatedPage<People> peoples =
elasticsearchTemplate.queryForPage(searchQuery, People.class, new
```

```
SearchResultMapper() {
            @Override
            public <T> AggregatedPage<T> mapResults(SearchResponse
searchResponse, Class<T> aClass, Pageable pageable) {
                /*
                {
  "took" : 190,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.7918377,
    "hits" : [
      {
        "_index" : "people_index",
        "_type" : "people_type",
        "_id" : "1",
        "_score" : 1.7918377,
        "_source" : {
          "id" : "1",
          "name" : "张三",
          "address" : "北京市回龙观东大街",
          "age" : 21
        },
        "highlight" : {
          "address" : [
            "<span>北京</span><span>市</span>回龙观东大街"
          ]
        }
      },
      {
        "_index" : "people_index",
        "_type" : "people_type",
        "_id" : "2",
        "_score" : 1.463562,
        "_source" : {
          "id" : "2",
          "name" : "李四",
          "address" : "北京市大兴区科创十四街",
```

```
          "age" : 22
        },
        "highlight" : {
          "address" : [
            "<span>北京</span><span>市</span>大兴区科创十四街"
          ]
        }
      },
      {
        "_index" : "people_index",
        "_type" : "people_type",
        "_id" : "3",
        "_score" : 0.38845786,
        "_source" : {
          "id" : "3",
          "name" : "王五",
          "address" : "天津市北京大街",
          "age" : 23
        },
        "highlight" : {
          "address" : [
            "天津市<span>北京</span>大街"
          ]
        }
      }
    ]
  }
}

                */

                // 取最里面层的hits
                SearchHit[] searchHits =
searchResponse.getHits().getHits();
                // 最终返回的数据
                List<T> peopleList = new ArrayList<>();
                for (SearchHit searchHit : searchHits) {
                    //需要把SearchHit转换为People
                    People peo = new People();
                    // 取出非高亮数据
                    Map<String, Object> mapSource =
searchHit.getSourceAsMap();
                    // 取出高亮数据
                    Map<String, HighlightField> mapHL =
```

```
searchHit.getHighlightFields();
                    // 把非高亮数据进行填充
                    peo.setId(mapSource.get("id").toString());
                    peo.setName(mapSource.get("name").toString());

peo.setAge(Integer.parseInt(mapSource.get("id").toString()));
                    // 判断是否有高亮，如果只有一个搜索条件，一定有这个高亮数据，
if 可以省略
                    if (mapHL.containsKey("address")) {
                        // 设置高亮数据

peo.setAddress(mapHL.get("address").getFragments()[0].toString());
                    }
                    // 把 people 添加到集合中
                    peopleList.add((T) peo);
                }
                // 如果没有分页，只有第一个参数。
                // 总条数在第一个 hits 里面。
                return new AggregatedPageImpl<>(peopleList, pageable,
searchResponse.getHits().totalHits);
            }

            @Override
            public <T> T mapSearchHit(SearchHit searchHit, Class<T> aClass)
{
                return null;
            }
        });

        // 通过 getContent 查询出最终 List<People>
        for (People people : peoples.getContent()) {
            System.out.println(people);
        }
    }
```