

4-8

第 4-8 课：Spring Boot 集成 Elasticsearch

ElasticSearch 是一个开源的搜索引擎，建立在一个全文搜索引擎库 Apache Lucene™ 基础之上。Lucene 可以说是当下最先进、高性能、全功能的搜索引擎库——无论是开源还是私有。

ElasticSearch 使用 Java 编写的，它的内部使用的是 Lucene 做索引与搜索，它的目的是使全文检索变得简单，通过隐藏 Lucene 的复杂性，取而代之提供了一套简单一致的 RESTful API。

然而，ElasticSearch 不仅仅是 Lucene，并且也不仅仅只是一个全文搜索引擎，它可以被下面这样准确地形容：

- 一个分布式的实时文档存储，每个字段可以被索引与搜索
- 一个分布式实时分析搜索引擎
- 能胜任上百个服务节点的扩展，并支持 PB 级别的结构化或者非结构化数据

ElasticSearch 已经被各大互联网公司验证其抢到的检索能力：

- *Wikipedia* 使用 Elasticsearch 提供带有高亮片段的全文搜索，还有 search-as-you-type 和 did-you-mean 的建议；
- 《卫报》使用 Elasticsearch 将网络社交数据结合到访客日志中，实时的给编辑们提供公众对于新文章的反馈；
- Stack Overflow 将地理位置查询融入全文检索中去，并且使用 more-like-this 接口去查找相关的问题与答案；
- GitHub 使用 Elasticsearch 对 1300 亿行代码进行查询。

小故事

关于 Elasticsearch 有一个小故事，在这里也分享给大家：

多年前，Shay Banon 是一位刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本的 Lucene。

直接基于 Lucene 工作会比较困难，因此 Shay 开始抽象 Lucene 代码以便 Java 程序员可以在应用中添加搜索功能，他发布了他的第一个开源项目，叫做“Compass”。

后来 Shay 找到了一份工作，这份工作处在高性能和内存数据网络的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的，然后他决定重写 Compass 库使其成为一个独立的服务叫做 Elasticsearch。

第一个公开版本出现在 2010 年 2 月，在那之后 Elasticsearch 已经成为 GitHub 上最受欢迎的项目之一，代码贡献者超过 300 人。一家主营 Elastic - Search 的公司就此成立，他们一边提供商业支持一边开发新功能，不过 ElasticSearch 将永远开源且对所有人可用。

Shay 的妻子依旧等待着她的食谱搜索……

在没有 Spring Boot 之前 Java 程序员使用 Elasticsearch 非常痛苦，需要对接链接资源、进行一系列的封装等操作。Spring Boot 在 spring-data-elasticsearch 的基础上进行了封装，让 Spring Boot 项目非常方便的去操作 Elasticsearch，如果前面了解过 JPA 技术的话，会发现他的操作语法和 JPA 非常的类似。

值得注意的是，Spring Data Elasticsearch 和 Elasticsearch 是有对应关系的，不同的版本之间不兼容，Spring Boot 2.1 对应的是 Spring Data Elasticsearch 3.1.2 版本。

Spring Data ElasticSearch	ElasticSearch
3.1.x	6.2.2
3.0.x	5.5.0
2.1.x	2.4.0
2.0.x	2.2.0
1.3.x	1.5.2

Spring Boot 集成 ElasticSearch

相关配置

在 Pom 中添加 ElasticSearch 的依赖:

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

配置 ElasticSearch 集群地址:

```
# 集群名(默认值: elasticsearch, 配置文件`cluster.name`: es-mongodb)  
spring.data.elasticsearch.cluster-name=es-mongodb  
# 集群节点地址列表, 用逗号分隔  
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

相关配置

```

@Document(indexName = "customer", type = "customer", shards = 1, replicas = 0, refreshInterval = "-1")
public class Customer {
    @Id
    private String id;
    private String userName;
    private String address;
    private int age;
    //省略部分 getter/setter
}

```

- @Document 注解会对实体中的所有属性建立索引
- indexName = "customer" 表示创建一个名称为 "customer" 的索引
- type = "customer" 表示在索引中创建一个名为 "customer" 的 type
- shards = 1 表示只使用一个分片
- replicas = 0 表示不使用复制
- refreshInterval = "-1" 表示禁用索引刷新

创建操作的 repository

```

public interface CustomerRepository extends ElasticsearchRepository
<Customer, String> {
    public List<Customer> findByAddress(String address);
    public Customer findByUserName(String userName);
    public int deleteByUserName(String userName);
}

```

我们创建了两个查询和一个删除的方法，从语法可以看出和前面 JPA 的使用方法非常类似，跟踪 ElasticsearchRepository 的代码会发现：

ElasticsearchRepository 继承于 ElasticsearchCrudRepository：

```

public interface ElasticsearchRepository<T, ID extends Serializable
> extends ElasticsearchCrudRepository<T, ID> {...}

```

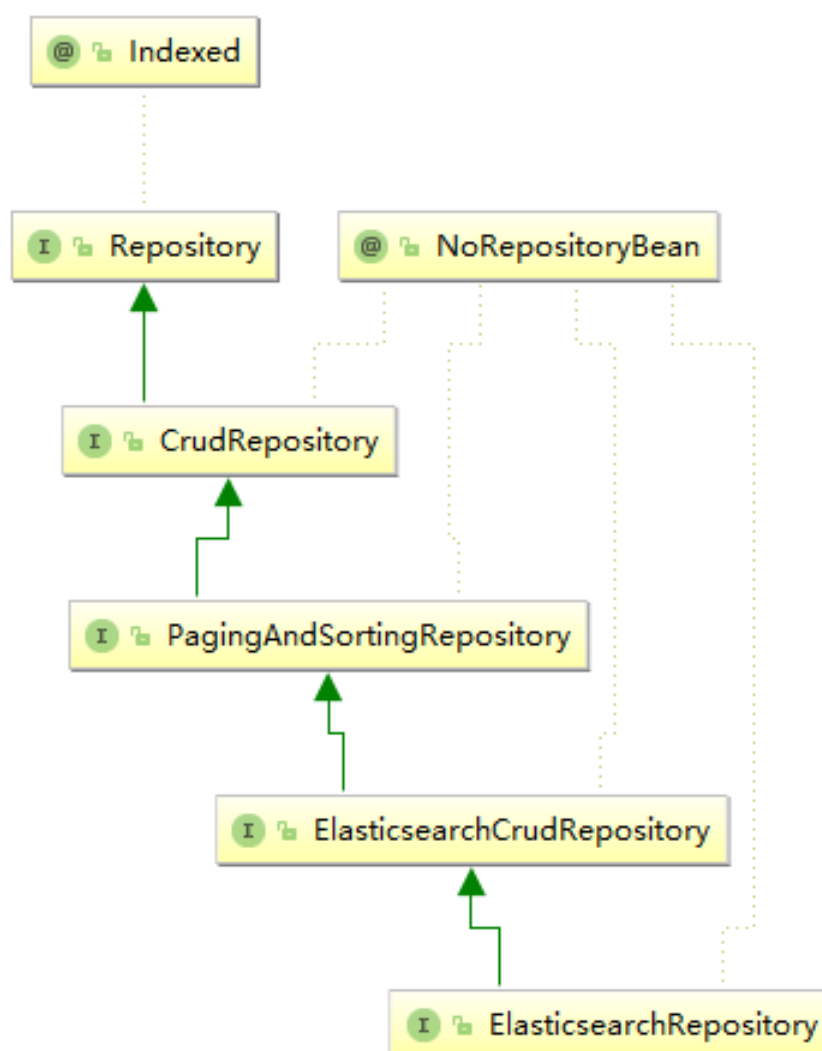
而 ElasticsearchCrudRepository 继承于 PagingAndSortingRepository：

```
public interface ElasticsearchCrudRepository<T, ID extends Serializable> extends PagingAndSortingRepository<T, ID>{...}
```

最后 PagingAndSortingRepository 继承于 CrudRepository:

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID>{...}
```

类图如下:



通过查看源码发现，ElasticsearchRepository 最终使用和 JPA 操作数据库使用的父类是一样的。通过这些也可以发现，Spring Data 项目中的成员在最上层有着统一的接口标准，只是在最终的实现层对不同的数据库进行了差异化封装。

以上简单配置完成之后我们在业务中就可以使用 ElasticSearch 了。

测试 CustomerRepository

创建一个测试类引入 CustomerRepository:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class CustomerRepositoryTest {
    @Autowired
    private CustomerRepository repository;
}
```

做一个数据插入测试：

```
@Test
public void saveCustomers() {
    repository.save(new Customer("Alice", "北京",13));
    repository.save(new Customer("Bob", "北京",23));
    repository.save(new Customer("neo", "西安",30));
    repository.save(new Customer("summer", "烟台",22));
}
```

repository 已经帮我们默认实现了很多的方法，其中就包括 save()。

我们对插入的数据做一个查询：

```
@Test
public void fetchAllCustomers() {
    System.out.println("Customers found with findAll():");
    System.out.println("-----");
    for (Customer customer : repository.findAll()) {
        System.out.println(customer);
    }
}
```

输出：

Customers found with **findAll()**:

```
Customer{id='aBVS7WYB8U8_i9prF8qm', userName='Alice', address='北京', age=13}
Customer{id='aRVS7WYB8U8_i9prF8rw', userName='Bob', address='北京', age=23}
Customer{id='ahVS7WYB8U8_i9prGMot', userName='neo', address='西安', age=30}
Customer{id='axVS7WYB8U8_i9prGMp2', userName='summer', address='北京市海淀区西直门', age=22}
```

通过查询可以发现，插入时自动生成了 ID 信息。

对插入的数据进行删除：

```
@Test
public void fetchAllCustomers() {
    @Test
    public void deleteCustomers() {
        repository.deleteAll();
        repository.deleteByUserName("neo");
    }
}
```

可以根据属性条件来删除，也可以全部删除。

对属性进行修改：

```
@Test
public void updateCustomers() {
    Customer customer= repository.findByUserName("summer");
    System.out.println(customer);
    customer.setAddress("北京市海淀区西直门");
    repository.save(customer);
    Customer xcustomer=repository.findByUserName("summer");
    System.out.println(xcustomer);
}
```

输出：

```
Customer[id=AWKVYFY4vPQX0UVGnJ7o, userName='summer', address='烟台']  
Customer[id=AWKVYFY4vPQX0UVGnJ7o, userName='summer', address='北京市海淀区西直门']
```

通过输出发现 summer 用户的地址信息已经被变更。

我们可以根据地址信息来查询在北京的顾客信息:

```
@Test  
public void fetchIndividualCustomers() {  
    for (Customer customer : repository.findByAddress("北京")) {  
        System.out.println(customer);  
    }  
}
```

输出:

```
Customer{id='aBVS7WYB8U8_i9prF8qm', userName='Alice', address='北京', age=13}  
Customer{id='aRVS7WYB8U8_i9prF8rw', userName='Bob', address='北京', age=23}  
Customer{id='axVS7WYB8U8_i9prGMp2', userName='summer', address='北京市海淀区西直门', age=22}
```

通过输出可以发现 Elasticsearch 默认给我们进行的就是字段全文（模糊）查询。

通过以上的示例发现使用 Spring Boot 操作 Elasticsearch 非常简单，通过少量代码即可实现我们日常大部分的业务需求。

高级使用

上面演示了在 Spring Boot 项目中对 Elasticsearch 的增、删、改、查操作，通过上面的操作也可以发现操作 Elasticsearch 的语法和 Spring Data JPA 的语法非常类似，下面介绍一些复杂的使用场景。

分页查询

分页查询有两种实现方式，第一种是使用 Spring Data 自带的分页方案，另一种是自行组织查询条件最后封装进行查询。我们先来看第一个方案：

```
@Test
public void fetchPageCustomers() {
    Sort sort = new Sort(Sort.Direction.DISC, "address.keyword");
    Pageable pageable = PageRequest.of(0, 10, sort);
    Page<Customer> customers=repository.findByAddress("北京", pageable);
    System.out.println("Page customers "+customers.getContent().toString());
}
```

这段代码的含义是，分页查询地址包含“北京”的客户信息，并且按照地址进行排序，每页显示 10 条。需要注意的是排序是使用的关键字是 address.keyword，而不是 address，属性后面带 .keyword 代表了精确匹配。

QueryBuilder

我们也可以使用 QueryBuilder 来构建分页查询，QueryBuilder 是一个功能强大的多条件查询构建工具，可以使用 QueryBuilder 构建出各种各样的查询条件。

```
@Test
public void fetchPage2Customers() {
    QueryBuilder customerQuery = QueryBuilders.boolQuery()
        .must(QueryBuilders.matchQuery("address", "北京"));
    Page<Customer> page = repository.search(customerQuery, PageRequest.of(0, 10));
    System.out.println("Page customers "+page.getContent().toString());
}
```

使用 QueryBuilder 可以构建多条件查询，再结合 PageRequest 最后使用 search() 方法完成分页查询。BoolQueryBuilder 有一些关键字和 AND、OR、NOT 一一对应：

- must(QueryBuilders):AND
- mustNot(QueryBuilders):NOT
- should::OR

QueryBuilder 是一个强大的多条件构建工具，有以下几种用法。

精确查询

单个匹配:

```
//不分词查询 参数1: 字段名, 参数2: 字段查询值, 因为不分词, 所以汉字只能查询一个字, 英语是一个单词
QueryBuilder queryBuilder=QueryBuilders.termQuery("fieldName", "fieldlValue");
//分词查询, 采用默认的分词器
QueryBuilder queryBuilder2 = QueryBuilders.matchQuery("fieldName", "fieldlValue");
```

多个匹配:

```
//不分词查询, 参数1: 字段名, 参数2: 多个字段查询值, 因为不分词, 因此汉字只能查询一个字, 英语是一个单词
QueryBuilder queryBuilder=QueryBuilders.termsQuery("fieldName", "fieldlValue1","fieldlValue2...");
//分词查询, 采用默认的分词器
QueryBuilder queryBuilder= QueryBuilders.multiMatchQuery("fieldl-Value", "fieldName1", "fieldName2", "fieldName3");
//匹配所有文件, 相当于就没有设置查询条件
QueryBuilder queryBuilder=QueryBuilders.matchAllQuery();
```

模糊查询

模糊查询常见的 5 个方法如下:

//1.常用的字符串查询

```
QueryBuilders.queryStringQuery("fieldValue").field("fieldName");//  
左右模糊
```

//2.常用的用于推荐相似内容的查询

```
QueryBuilders.moreLikeThisQuery(new String[] {"fieldName"}).add-  
LikeText("pipeidhua");//如果不指定filed-  
Name, 则默认全部, 常用在相似内容的推荐上
```

//3.前缀查询, 如果字段没分词, 就匹配整个字段前缀

```
QueryBuilders.prefixQuery("fieldName","fieldValue");
```

//4.fuzzy query:分词模糊查询, 通过增加 fuzziness

模糊属性来查询, 如能够匹配 hotel-

Name 为 tel 前或后加一个字母的文档, fuzziness

的含义是检索的 term 前后增加或减少 n 个单词的匹配查询

```
QueryBuilders.fuzzyQuery("hotelName", "tel").fuzziness(Fuzziness.ONE);
```

//5.wildcard query:通配符查询, 支持* 任意字符串; ? 任意一个字符

```
QueryBuilders.wildcardQuery("fieldName","ctr*");//前面是field-  
name, 后面是带匹配字符的字符串
```

```
QueryBuilders.wildcardQuery("fieldName","c?r?");
```

范围查询

//闭区间查询

```
QueryBuilder queryBuilder0 = QueryBuilders.rangeQuery("fieldName").  
from("fieldValue1").to("fieldValue2");
```

//开区间查询

```
QueryBuilder queryBuilder1 = QueryBuilders.rangeQuery("fieldName").  
from("fieldValue1").to("fieldValue2").includeUpper(false).include-  
Lower(false);//默认是 true, 也就是包含
```

//大于

```
QueryBuilder queryBuilder2 = QueryBuilders.rangeQuery("fieldName").  
gt("fieldValue");
```

//大于等于

```
QueryBuilder queryBuilder3 = QueryBuilders.rangeQuery("fieldName").  
gte("fieldValue");
```

//小于

```
QueryBuilder queryBuilder4 = QueryBuilders.rangeQuery("fieldName").  
lt("fieldValue");
```

//小于等于

```
QueryBuilder queryBuilder5 = QueryBuilders.rangeQuery("fieldName").  
lte("fieldValue");
```

多条件查询

```
QueryBuilders.boolQuery()  
QueryBuilders.boolQuery().must();//文档必须完全匹配条件, 相当于 and  
QueryBuilders.boolQuery().mustNot();//文档必须不匹配条件, 相当于 not
```

聚合查询

聚合查询分为五步来实现, 我们以统计客户总年龄为例进行演示。

第一步, 使用 QueryBuilder 构建查询条件:

```
QueryBuilder customerQuery = QueryBuilders.boolQuery()  
    .must(QueryBuilders.matchQuery("address", "北京"));
```

第二步, 使用 SumAggregationBuilder 指明需要聚合的字段:

```
SumAggregationBuilder sumBuilder = AggregationBuilders.sum("sumAge"  
    ).field("age");
```

第三步, 以前两部分的内容为参数构建成 SearchQuery:

```
SearchQuery searchQuery = new NativeSearchQueryBuilder()  
    .withQuery(customerQuery)  
    .addAggregation(sumBuilder)  
    .build();
```

第四步, 使用 Aggregations 进行查询:

```
Aggregations aggregations = elasticsearchTemplate.query(search-  
Query, new ResultsExtractor<Aggregations>() {  
    @Override  
    public Aggregations extract(SearchResponse response) {  
        return response.getAggregations();  
    }  
});
```

第五步, 解析聚合查询结果:


```
//转换成 map 集合
Map<String, Aggregation> aggregationMap = aggregations.asMap();
//获得对应的聚合函数的聚合子类，该聚合子类也是个 map 集合，里面的 value 就是桶 Bucket，我们要获得 Bucket
InternalSum sumAge = (InternalSum) aggregationMap.get("sumAge");
System.out.println("sum age is "+sumAge.getValue());
```

以上就是聚合查询的使用方式。

总结

Spring Boot 对 Elasticsearch 的集成延续了 Spring Data 的思想，通过继承对应的 repository 默认帮我们实现了很多常用操作，通过注解也非常的方便设置索引映射在 Elasticsearch 的数据使用。在大规模搜索中使用 Spring Boot 操作 Elasticsearch 是一个最佳的选择。

点击这里下载源码 (https://github.com/ityouknow/spring-boot-learning/tree/gitbook_column2.0)。

 (https://gitee.com/ityouknow/spring-boot-learning/tree/gitbook_column2.0)