EsClientRHL

EsClientRHL是一个可基于springboot的elasticsearch RestHighLevelClient客户端调用封装工具，主要提供了es索引结构工具、es索引数据增删改工具、es查询工具、es数据分析工具。由于采用RestHighLevelClient，所以版本兼容问题应该能得到一定改善。

选择EsClientRHL原因

- 目前spring-data-elasticsearch底层采用es官方TransportClient，而es官方计划放弃TransportClient，工具以es官方推荐的RestHighLevelClient进行封装
- spring-data-elasticsearch支持的api有限，而EsClientRHL支持更丰富的api调用
- 能够极大简化java client API，并不断更新，让es更高级的功能更轻松的使用
- 支持两种自动化的功能，减轻开发者工作量，使其更专注于业务开发

1. 支持启动自动扫描elasticsearch索引实体类，并为没有索引结构的实体自动创建索引结构
2. 支持开发者只定义一个接口，就拥有了常用与es交互的黑魔法

- 组件中包含了：es索引数据增删改、es查询、es数据分析等丰富的API工具，开发者可以通过EsClientRHL来参考在java中如何与elasticsearch进行各种交互
- EsClientRHL中部分API结合了实际场景中最佳实践的使用方法
- 总之ESClientRHL能给您带来帮助，那它就有存在的价值，如果对您有些许帮助，请不吝Star https://gitee.com/zxporz/ESClientRHL

使用前你应该具有哪些技能

- springboot
- maven
- elasticsearch基础概念
- elasticsearch rest api含义以及用法

工具功能范围介绍

索引结构管理

- 判断索引是否存在
- 索引结构创建
- 自动定制索引结构mapping
- 删除索引结构

CRUD

- LowLevelClient查询
- 新增索引数据
- 批量新增索引数据
- 分批次新增索引数据
- 覆盖更新索引数据
- 部分更新索引数据
- 批量更新索引
- 分批次批量更新索引
- 删除索引数据
- 判断索引数据是否存在
- 原生查询
- 支持uri query string的查询
- 支持sql查询
- 支持查询条件的定制查询
- 支持查询条件+最大返回条数的定制查询
- 支持分页、高亮、排序、查询条件的定制查询
- 高级查询（支持分页、高亮、排序、路由、指定结果字段、查询条件、search after）
- count查询
- scroll查询（用于大数据量查询）
- 模版查询-保存模版
- 模版查询-注册模版
- 模版查询-内联模版
- 搜索建议completion suggest
- 搜索建议phrase suggest
- 根据ID查询
- mget查询

- 按照多索引查询说明（2019-03-19新增）

聚合查询

- 原生聚合查询
- 普通聚合查询
- 分组普通聚合查询
- 下钻（2层）聚合查询
- 统计聚合查询
- 分组统计聚合查询
- 基数查询
- 百分比聚合查询
- 百分等级聚合查询
- 过滤器聚合查询
- 直方图聚合查询
- 日期直方图聚合查询

工具源码结构介绍

annotation

存放一些注解，用于简化组件使用

config

基于springboot的自动化的功能，包括自动配置es客户端组件以及自动管理索引结构的功能

enums

基础数据的枚举

index

索引结构管理的功能包

repository

CURD+聚合的功能包

util

内部工具包

开始使用

使用前说明

- 组件基于JDK1.8编译，请注意JDK版本的选择
- 目前只支持springboot方式集成，如果需要与普通spring系统集成需要做简单改造
- 如果您并不需要使用组件自带的集成方式，仅需参考与es集成的方式请注意关注`ElasticsearchTemplateImpl`类中的代码即可

maven依赖

请把组件安装到maven仓库

```
<dependency>
    <groupId>org.zxp</groupId>
    <artifactId>esclientrhl</artifactId>
    <version>7.0.0</version>
</dependency>
```

特别注意：建议在引入的springboot工程中pom文件添加elasticsearch版本号，否则可能被springboot parent工程覆盖。

```
<properties>
……
<elasticsearch.version>7.3.1</elasticsearch.version>
……
</properties>
```

引入组件

添加EnableESTools注解

在springboot启动类上添加

`@EnableESTools`注解能够帮助开发人员自动注入工具服务，简化配置，并引入自动发现es索引结构实体类的功能，识别ESCRepository接口并自动生成代理的功能

EnableESTools注解用法：
用法1：最全配置

```
@EnableESTools(
basePackages = {"ESCRepository接口包路径数组"},
entityPath = {"实体类包路径数组"},
printregmsg = true)//详见运维功能部分
```

用法2：最简配置
默认扫描启动类所在包下的所有类

```
@EnableESTools
```

用法3：只扫描实体类路径

```
@EnableESTools({"实体类包路径数组"})
```

将@EnableESTools配置于SpringBoot启动类上

```
@SpringBootApplication
@EnableESTools
public class EsdemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(EsdemoApplication.class, args);
    }
}
```

application.properties添加elasticsearch服务的uri

application.properties配置elasticsearch服务的uri，如果有多个（集群情况）请用半角逗号`,`隔开，必须配置

```
elasticsearch.host=127.0.0.1:9200
```

application.properties添加elasticsearch用户名密码（基于xpack仅支持es7+的版本）

没有可以不用配置

```
elasticsearch.username=elastic
elasticsearch.password=changeme
```

使用组件

在spring管理的bean内直接自动注入组件内置的两个工具服务：`ElasticsearchTemplate`、`ElasticsearchIndex`并调用相关api即可

```
@Service
public class CompletionSuggestServiceImpl implements CompletionSuggestService {
    @Autowired
    ElasticsearchTemplate<Book,String> elasticsearchTemplate;
```

如果只想引入一些简单的基础功能，推荐采用接口代理的方式，使用将会更加简便
定义接口，只需要继承ESCRepository接口即可

```
public interface Main2Repository<Main2,String> extends
ESCRepository<Main2,String> {
}
```

使用时，只需要将定义的接口引入即可

```
@Autowired
Main2Repository main2Repository;
```

使用方法与ElasticsearchTemplate大同小异，只有一些比较基础的方法，去掉了Clazz类类型的入参

```
Main2 main2 = new Main2();
main2.setProposal_no("qq123549440");
main2.setBusiness_nature_name("渠道");
main2.setAppli_name("esclientrhl");
main2Repository.save(main2);
System.out.println(main2Repository.getById("22222"));

Map map2 = main2Repository.aggs("proposal_no",
AggsType.count,null,"appli_name");
map2.forEach((o, o2) -> System.out.println(o + "=====" + o2));
```

注意事项：
如果采用自动代理接口的方式，需要注意以下几点：

1. 接口必须继承自ESCRepository，并且定义接口时必须注明泛型的真实类型
2. 对应的实体类必须添加ESMetaData注解，组件才能自动识别
3. 实体类名称整个工程内不能重复，否则会导致生成代理类失败

索引管理功能

元数据配置

用于定制es索引结构对应实体类的元数据

```
@ESMetaData(indexName = "index", number_of_shards = 5,number_of_replicas = 0)
```

包含的主要配置信息以及默认值如下

```
/**
 * 检索时的索引名称，如果不配置则默认为和indexName一致，该注解项仅支持搜索
 * 并不建议这么做，建议通过特定方法来做跨索引查询
 */
String[] searchIndexNames() default {};
/**
 * 索引名称，必须配置
 */
String indexName();
/**
 * 索引类型，可以不配置，不配置默认为_doc，墙裂建议每个index下只有一个type
 */
String indexType() default "";
/**
 * 主分片数量
 */
int number_of_shards() default 5;
/**
 * 备份分片数量
 */
int number_of_replicas() default 1;
/**
 * 是否打印日志
 * @return
 */
boolean printLog() default false;
```

索引结构配置

用于定制es索引结构对应实体类的索引结构，以简化创建索引工作。将相关注解配置于实体类field上，用于标识field对应elasticsearch索引结构字段的相关信息

```
@ESID
private String proposal_no;
@ESMapping(datatype = DataType.keyword_type)
private String risk_code;
@ESMapping(datatype = DataType.text_type)
private String risk_name;
```

`@ESID`标识es主键（自动对应es索引数据_id字段），注意：主键的类型需要与`ElasticsearchTemplate`的第二泛型一致

`@ESMapping`标识字段对应es索引结构字段的相关信息

```
/**
 * 数据类型（包含 关键字类型）
```

```
 */
DataType datatype() default DataType.text_type;
/**
 * 间接关键字
 */
boolean keyword() default true;
/**
 * 关键字忽略字数
 */
int ignore_above() default 256;
/**
 * 是否支持ngram，高效全文搜索提示（定制gram分词器，请参照官方例
https://www.elastic.co/guide/en/elasticsearch/reference/7.x/analysis-ngram-
tokenizer.html）
 */
boolean ngram() default false;
/**
 * 是否支持suggest，高效前缀搜索提示
 */
boolean suggest() default false;
/**
 * 索引分词器设置（研究类型）
 */
Analyzer analyzer() default Analyzer.standard;
/**
 * 搜索内容分词器设置
 */
Analyzer search_analyzer() default Analyzer.standard;
/**
 * 是否允许被搜索
 */
boolean allow_search() default true;

/**
 * 拷贝到哪个字段，代替_all
 */
String copy_to() default "";

/**
* null_value指定，默认空字符串不会为mapping添加null_value
* 对于值是null的进行处理，当值为null是按照注解指定的‘null_value’值进行查询可以查到
* 需要注意的是要与根本没有某字段区分（没有某字段需要用Exists Query进行查询）
* 建议设置值为NULL_VALUE
* @return
*/
String null_value() default "";

/**
* nested对应的类型，默认为Object.Class。
* 对于DataType是nested_type的类型才需要添加的注解，通过这个注解生成嵌套类型的索引
* 例如：
* @ESMapping(datatype = DataType.nested_type, nested_class = EsFundDto.class)
*
* @return
*/
Class nested_class() default Object.class;
```

如果对字段类型要求没有那么高，则不配置，组件可以支持自动适配mapping

根据配置信息自动创建索引结构mapping

项目启动时，组件会自动识别es实体类上配置的`@ESMetaData`注解，如果对应的索引结构没有创建，自动根据mapping注解配置创建相关索引结构。

如果实体类不在启动类的包路径下，如需启用此功能，需要在启动注解上配置实体类路径。

```
@EnableESTools(entityPath = "com.*.esdemo.domain")
```

针对nested类型支持的说明

包含有nested成员变量的索引不支持部分字段更新功能（支持覆盖更新）
nested对象中的变量不支持聚合查询
支持自动创建nested注解创建mapping但目前nested只支持一层nested并且不能支持复杂的mapping配置（只支持对type的配置）

配置示例（下面是list方式，也支持非list方式），datatype除了需要指定为nested_type枚举类型外，还需要指定nested类类型

```
@ESMapping(datatype = DataType.nested_type,nested_class = Actors.class)
private List<Actors> actors;
```

Actors中配置`@ESMapping`即可

```
public class Actors {
    @ESMapping
    private String first_name;
    @ESMapping
    private String last_name;
```

ngram、completion suggest、phrase suggest的区别

ngram

- ngram需要定制setting和mapping
- 仍然利用倒排索引，将单词中字母或字进位组成倒排索引（edge ngram），可以高效前缀检索，可以从中间的词进行检索
- 检索时采用match_phrase即可

completion suggest

- completion suggest需要定制mapping
- 没有使用倒排索引，而是使用了倒排索引词典FST数据结构并缓存与内存，有非常好的性能，只能严格前缀检索，不支持中间的词检索
- 检索需要单独的api

phrase suggest

- phrase suggest不需要定制
- 没有completion suggest性能好，侧重点是匹配相似的词（输入错误的词），根据相对模糊的输入词条给出搜索建议以提供更好的搜索体验和准备下一步检索提供基础
- 检索需要单独的api

手工创建或删除索引结构

```
elasticsearchIndex.dropIndex(Main2.class);
elasticsearchIndex.createIndex(Main2.class);
```

判断索引是否存在

```
/**
 * 索引是否存在
 * @param clazz传入es索引结构对应实体类
 * @throws Exception
 */
public boolean exists(Class<T> clazz) throws Exception;
```

```
elasticsearchIndex.exists(Main2.class)
```

CRUD功能说明

LowLevelClient查询

这种情况通常适用于直接发JSON报文查询或操作es服务端，本方法并没有做太多的封装，基本保留了原生的出入参

```
//自动注入工具类
@Autowired
ElasticsearchTemplate elasticsearchTemplate;
//执行查询
Request request = new Request("GET","/esdemo/_search");
request.setEntity(new NStringEntity("{\"query\":{\"match_all\":
{\"boost\":1.0}}}",ContentType.APPLICATION_JSON));
Response response = elasticsearchTemplate.request(request);
RequestLine requestLine = response.getRequestLine();
HttpHost host = response.getHost();
int statusCode = response.getStatusLine().getStatusCode();
Header[] headers = response.getHeaders();
String responseBody = EntityUtils.toString(response.getEntity());
System.out.println(responseBody);
```

新增索引数据

如无特殊说明下面查询都默认注入了工具类，工具类第一泛型是要操作的es索引结构的类，第二泛型是索引对应主键的类

```
@Autowired
ElasticsearchTemplate<Main2,String> elasticsearchTemplate;
```

Main2类型的主键是String

```
@ESMetaData(indexName = "index", number_of_shards = 5,number_of_replicas = 0)
public class Main2 implements Serializable {
    private static final long serialVersionUID = 1L;
    @ESID
    private String proposal_no;
```

```
Main2 main = new Main2();
main.setProposal_no("main2");
main.setAppli_code("123");
main.setAppli_name("456");
elasticsearchTemplate.save(main);
```

可以定制`routing`路由字段来指定数据被索引到哪个分片上

```
Main2 main2 = new Main2();
main2.setProposal_no("qq360");
main2.setAppli_name("zzxxpp");
elasticsearchTemplate.save(main2, "R01");
```

批量新增索引数据

```
List<Main2> list = new ArrayList<>();
Main2 main1 = new Main2();
main1.setProposal_no("main1");
main1.setAppli_code("123");
main1.setAppli_name("456");
Main2 main2 = new Main2();
main2.setProposal_no("main2");
main2.setAppli_code("123");
main2.setAppli_name("456");
Main2 main3 = new Main2();
main3.setProposal_no("main3");
main3.setAppli_code("123");
main3.setAppli_name("456");
list.add(main1);
list.add(main2);
list.add(main3);
elasticsearchTemplate.save(list);
```

分批次新增索引数据

相比于批量新增索引数据，分批次新增索引数据考虑了es服务端批量索引数据的内存瓶颈，将根据一些简单的策略对传入的数据列表进行拆分并顺序索引

您可以修改org.zxp.esclientrhl.util.Constant配置类中的如下变量，结合最佳实践，默认以5000条为一批

```
 //批量更新（新增）每批次条数
public static int BULK_COUNT = 5000;
```

分批次新增索引数据提供了一个新的方法：

```
Main2 main1 = new Main2();
main1.setProposal_no("aaa");
main1.setBusiness_nature_name("aaaaaa2");
Main2 main2 = new Main2();
main2.setProposal_no("bbb");
main2.setBusiness_nature_name("aaaaaa2");
Main2 main3 = new Main2();
main3.setProposal_no("ccc");
main3.setBusiness_nature_name("aaaaaa2");
Main2 main4 = new Main2();
main4.setProposal_no("ddd");
main4.setBusiness_nature_name("aaaaaa2");
elasticsearchTemplate.saveBatch(Arrays.asList(main1,main2,main3,main4));
```

部分更新索引数据

```
Main2 main1 = new Main2();
main1.setProposal_no("main1");
main1.setInsured_code("123");
elasticsearchTemplate.update(main1);
```

覆盖更新索引数据

```
Main2 main1 = new Main2();
main1.setProposal_no("main1");
main1.setInsured_code("123");
elasticsearchTemplate.updateCover(main1);
```

部分更新相比于覆盖更新的区别是，部分更新只会更新set了的字段值

批量更新索引

```
/**
* 根据queryBuilder所查结果，按照有值字段更新索引
*
* @param queryBuilder
* @param t    满足queryBuilder查询出条件的结果集更新为t的有值字段值
* @param clazz
* @param limitcount 更新字段不能超出limitcount
* @param asyn true异步处理   否则同步处理
* @return
* @throws Exception
*/
public BulkResponse batchUpdate(QueryBuilder queryBuilder, T t, Class clazz, int
limitcount, boolean asyn) throws Exception;
```

```
//将appli_name是123的结果集每条数据的sum_amount更新为1000，异步更新，更新条数不得超过30条
Main2 main1 = new Main2();
main1.setSum_amount(1000);
elasticsearchTemplate.batchUpdate(QueryBuilders.matchQuery("appli_name","123"),m
ain1,Main2.class,30, true);
```

批量更新索引不支持覆盖更新

分批次批量更新索引

调整分批次策略详见“分批次新增索引数据”章节的内容

```
Main2 main1 = new Main2();
main1.setProposal_no("aaa");
main1.setBusiness_nature_name("aaaaaa2");
Main2 main2 = new Main2();
main2.setProposal_no("bbb");
main2.setBusiness_nature_name("aaaaaa2");
Main2 main3 = new Main2();
main3.setProposal_no("ccc");
main3.setBusiness_nature_name("aaaaaa2");
Main2 main4 = new Main2();
main4.setProposal_no("ddd");
main4.setBusiness_nature_name("aaaaaa2");
elasticsearchTemplate.bulkUpdateBatch(Arrays.asList(main1,main2,main3,main4));
```

删除索引数据

```
//通过对象删除，ID必须有值
Main2 main1 = new Main2();
main1.setProposal_no("main1");
main1.setInsured_code("123");
elasticsearchTemplate.delete(main1);
//通过ID删除
elasticsearchTemplate.deleteById("main1",Main2.class);
```

定制 `routing` 路由信息删除

```
//只有routing信息正确时才能成功删除
elasticsearchTemplate.delete(main2,"RO2");
//不管索引数据是否指定了routing都能删除
elasticsearchTemplate.delete(main2);
```

根据查询条件删除索引数据

```
elasticsearchTemplate.deleteByCondition(QueryBuilders.matchQuery("appli_name","2
"),Main5.class);
```

判断索引数据是否存在

```
Main2 main1 = new Main2();
main1.setProposal_no("main1");
main1.setInsured_code("123");
boolean exists = elasticsearchTemplate.exists("main1",Main2.class);
System.out.println(exists);
```

原生查询

searchRequest是官方原生查询输入，此方法在工具无法满足需求时使用

```
SearchRequest searchRequest = new SearchRequest(new String[]{"index"});
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
searchSourceBuilder.query(new MatchAllQueryBuilder());
```

```
searchSourceBuilder.from(0);
searchSourceBuilder.size(10);
searchRequest.source(searchSourceBuilder);
SearchResponse searchResponse = elasticsearchTemplate.search(searchRequest);

SearchHits hits = searchResponse.getHits();
SearchHit[] searchHits = hits.getHits();
for (SearchHit hit : searchHits) {
    Main2 t = JsonUtils.string2Obj(hit.getSourceAsString(), Main2.class);
    System.out.println(t);
}
```

支持uri query string的查询

以uri+参数的方式（query string）查询并返回结果
api用法参考官方文档：
https://www.elastic.co/guide/en/elasticsearch/reference/7.0/search-uri-request.html

```
//"q=aaa"查询字段中包含aaa匹配的结果
List<Main2> list = elasticsearchTemplate.searchUri("q=aaa",Main2.class);
//"q=sum_premium:100"查询sum_premium为100的结果
List<Main2> list =
elasticsearchTemplate.searchUri("q=sum_premium:100",Main2.class);
```

注意：部分高级uri查询功能（如范围查询）可能会不可用

支持sql查询

将sql（支持mysql语法）语句传入方法并以各种形式（方法返回为文本）返回的的查询方法
值得注意的是该方法不支持自动识别索引，需要将from后的索引名称定义正确（区分大小写）
另外该方法亦不支持自动泛型转化

```
//全查询
String result = elasticsearchTemplate.queryBySQL("SELECT * FROM index ORDER BY
sum_premium DESC LIMIT 5", SqlFormat.TXT);
//查询count
String result = elasticsearchTemplate.queryBySQL("SELECT count(*) FROM index ",
SqlFormat.TXT);
//分组查询
String result = elasticsearchTemplate.queryBySQL("SELECT
risk_code,sum(sum_premium) FROM index group by risk_code", SqlFormat.TXT);
```

该方法第二个参数是返回结果的枚举类型，类型列表详见下文：

```
CSV("csv","text/csv"),
JSON("json","application/json"),
TSV("tsv","text/tab-separated-values"),
TXT("txt","text/plain"),
YAML("yaml","application/yaml"),
CBOR("cbor","application/cbor"),
SMILE("smile","application/smile");
```

支持查询条件的定制查询

```
/**
 * 非分页查询
 * 目前暂时传入类类型
 * @param queryBuilder 查询条件
 * @param clazz 泛型对应类类型
 * @return 泛型定义好的es索引结构实体类查询结果集合
 * @throws Exception
 */
public List<T> search(QueryBuilder queryBuilder, Class<T> clazz) throws
Exception;
```

```
//这里简单通过matchall（全查询）的方式进行演示
//QueryBuilder的用法会在单独章节介绍
List<Main2> main2List = elasticsearchTemplate.search(new
MatchAllQueryBuilder(),Main2.class);
main2List.forEach(main2 -> System.out.println(main2));
```

支持查询条件+最大返回条数的定制查询

```
/**
* 非分页查询，指定最大返回条数
* 目前暂时传入类类型
* @param queryBuilder
* @param limitSize 最大返回条数
* @param clazz
* @return
* @throws Exception
*/
public List<T> searchMore(QueryBuilder queryBuilder,int limitSize, Class<T>
clazz) throws Exception;
```

```
//最多返回7条数据
List<Main2> main2List = elasticsearchTemplate.searchMore(new
MatchAllQueryBuilder(),7,Main2.class);
System.out.println(main2List.size());
main2List.forEach(main2 -> System.out.println(main2));
```

代理接口用法：

```
//最多返回6条数据
List<Main2> ll = main2Repository.searchMore(new MatchAllQueryBuilder(), 6);
System.out.println(ll.size());
ll.forEach(s -> System.out.println(s));
```

支持分页、高亮、排序、查询条件的定制查询

```
/**
 * 支持分页、高亮、排序的查询
 * @param queryBuilder 查询条件
 * @param pageSortHighLight 封装了分页、排序、高亮的规格信息
 * @param clazz 泛型对应类类型
 * @return 泛型定义好的es索引结构实体类查询结果集合
 * @throws Exception
 */
public PageList<T> search(QueryBuilder queryBuilder, PageSortHighLight
pageSortHighLight, Class<T> clazz) throws Exception;
```

```
//定制分页信息
int currentPage = 1;
int pageSize = 10;
//分页
PageSortHighLight psh = new PageSortHighLight(currentPage,pageSize);
//排序字段，注意如果proposal_no是text类型会默认带有keyword性质，需要拼接.keyword
String sorter = "proposal_no.keyword";
Sort.Order order = new Sort.Order(SortOrder.ASC,sorter);
psh.setSort(new Sort(order));
//定制高亮，如果定制了高亮，返回结果会自动替换字段值为高亮内容
psh.setHighLight(new HighLight().field("risk_code"));
//可以单独定义高亮的格式
//new HighLight().setPreTag("<em>");
//new HighLight().setPostTag("</em>");
PageList<Main2> pageList = new PageList<>();
pageList = elasticsearchTemplate.search(new MatchAllQueryBuilder(), psh,
Main2.class);
pageList.getList().forEach(main2 -> System.out.println(main2));
```

高级查询

高级查询除了包含分页、排序、高亮的定制查询，还支持指定返回结果字段的定制，以及路由查询的指定

指定返回结果字段，将指定的字段（可以指定多个字段，数组形式）设定在PageSortHighLight对象中即可

```
PageSortHighLight psh = new PageSortHighLight(1, 50);
//返回结果只包含proposal_no字段
String[] includes = {"proposal_no"};
psh.setIncludes(includes);
List<Main2> list = elasticsearchTemplate.search(new MatchAllQueryBuilder(),psh,
Main2.class).getList();
list.forEach(s -> System.out.println(s));
}
```

还可以定制以下返回策略

```
//除了business_nature_name字段其余的返回
String[] excludes = {"business_nature_name"};
psh.setExcludes(excludes);
//返回以risk开头的字段
String[] includes = {"risk*"};
psh.setIncludes(includes);
```

指定路由名称进行查询

```
//索引数据时也必须索引到routing为R10的分片中才能查到
Attach attach = new Attach();
attach.setRouting("R01");
elasticsearchTemplate.search(QueryBuilders.termQuery("proposal_no", "qq360"),
attach, Main2.class)
.getList().forEach(s -> System.out.println(s));
```

search after查询

search after主要解决了elasticsearch深分页（deep paging）的问题，但只能从第一页一页一页向后翻
https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-body.html#request-body-search-search-after

```
Attach attach = new Attach();
//【必须】设置searchAfter模式
attach.setSearchAfter(true);
//【非必须】设置分页信息（如果不设置，默认一页10条数据）
PageSortHighLight pageSortHighLight = new PageSortHighLight(1, 10);
//【必须】设置排序字段
String sorter = "sum_amount";
Sort.Order order = new Sort.Order(SortOrder.ASC,sorter);
pageSortHighLight.setSort(new Sort(order));
attach.setPageSortHighLight(pageSortHighLight);
PageList page = elasticsearchTemplate.search(new
MatchAllQueryBuilder(),attach,Main2.class);
//获取第一页数据
page.getList().forEach(s -> System.out.println(s));
Object[] sortValues = page.getSortValues();
//分别获取每页数据
while (true) {
    //必须将前一次获取的sortValues设置正确
    attach.setSortValues(sortValues);
    page = elasticsearchTemplate.search(new
MatchAllQueryBuilder(),attach,Main2.class);
    //当没有数据说明已经遍历完成
    if (page.getList() != null && page.getList().size() != 0) {
        page.getList().forEach(s -> System.out.println(s));
        sortValues = page.getSortValues();
    } else {
        break;
    }
}
```

count查询

结合查询条件查询结果的数据量

```
long count = elasticsearchTemplate.count(new
MatchAllQueryBuilder(),Main2.class);
System.out.println(count);
```

scroll查询

为了防止数据量多大导致的内存溢出，将scroll方法拆分为多步执行

```
/**
* scroll方式查询，创建scroll
* @param queryBuilder
* @param clazz
* @param time
* @param size
* @return
* @throws Exception
*/
public ScrollResponse<T> createScroll(QueryBuilder queryBuilder, Class<T> clazz,
long time, int size) throws Exception;

/**
* scroll方式查询，创建scroll
* @param queryBuilder
* @param clazz
* @param time
* @param size
* @param indexs
* @return
* @throws Exception
*/
public ScrollResponse<T> createScroll(QueryBuilder queryBuilder, Class<T> clazz,
long time, int size , String... indexs) throws Exception;
```

```
//创建scroll并获得第一批数据
ScrollResponse<Main2> scrollResponse = elasticsearchTemplate.createScroll(new
MatchAllQueryBuilder(), Main2.class, 1, 100);
scrollResponse.getList().forEach(s -> System.out.println(s));
String scrollId = scrollResponse.getScrollId();
//通过scrollId获取其他批次的数据
while (true){
    scrollResponse = elasticsearchTemplate.queryScroll(Main2.class, 1,
scrollId);
    if(scrollResponse.getList() != null && scrollResponse.getList().size() != 0)
{
    scrollResponse.getList().forEach(s -> System.out.println(s));
    scrollId = scrollResponse.getScrollId();
}else{
    break;
}
```

以下两种方式为了避免内存溢出已经不建议使用

```
/**
 * scroll方式查询，用于大数据量查询
 * @param queryBuilder 查询条件
 * @param clazz 类类型
 * @param time 保留小时数
 * @return
 * @throws Exception
 */
@Deprecated
public List<T> scroll(QueryBuilder queryBuilder, Class<T> clazz, long time)
throws Exception;
```

```
/**
 * scroll方式查询，用于大数据量查询，默认了保留时间为2小时(Constant.DEFAULT_SCROLL_TIME)
 * @param queryBuilder
 * @param clazz
 * @return
 * @throws Exception
 */
@Deprecated
public List<T> scroll(QueryBuilder queryBuilder, Class<T> clazz) throws
Exception;
```

```
//默认scroll镜像保留2小时
List<Main2> main2List = elasticsearchTemplate.scroll(new
MatchAllQueryBuilder(),Main2.class);
main2List.forEach(main2 -> System.out.println(main2));

//指定scroll镜像保留5小时
//List<Main2> main2List = elasticsearchTemplate.scroll(new
MatchAllQueryBuilder(),Main2.class,5);
```

模版查询-保存模版

```
/**
* 保存Template
* @param templateName 模版名称
* @param templateSource 模版内容
* @return
*/
public Response saveTemplate(String templateName,String templateSource) throws
Exception;
```

保存一个名称为tempdemo1的模版，模版内容见变量templatesource值

```
String templatesource = "{\n" +
"  \"script\": {\n" +
"    \"lang\": \"mustache\",\n" +
"    \"source\": {\n" +
"      \"_source\": [\n" +
"        \"proposal_no\",\"appli_name\"\n" +
"      ],\n" +
"      \"size\": 20,\n" +
"      \"query\": {\n" +
"        \"term\": {\n" +
"          \"appli_name\": \"{{name}}\"\n" +
"        }\n" +
"      }\n" +
"    }\n" +
"  }\n" +
"}";
elasticsearchTemplate.saveTemplate("tempdemo1",templatesource);
```

模版查询-注册模版

```
/**
* Template方式搜索，Template已经保存在script目录下
* @param template_params 模版参数
* @param templateName 模版名称
* @param clazz
* @return
*/
public List<T> searchTemplate(Map<String, Object> template_params,String
templateName,Class<T> clazz) throws Exception;
```

套用查询上面保存的模版tempdemo1

```
Map param = new HashMap();
        param.put("name","123");

elasticsearchTemplate.searchTemplate(param,"tempdemo1",Main2.class).forEach(s ->
System.out.println(s));
```

模版查询-内联模版

```
/**
* Template方式搜索，Template内容以参数方式传入
* @param template_params 模版参数
* @param templateSource 模版内容
* @param clazz
* @return
*/
public List<T> searchTemplateBySource(Map<String, Object> template_params,String
templateSource,Class<T> clazz) throws Exception;
```

直接传模版内容进行查询

```
Map param = new HashMap();
param.put("name","123");
String templatesource = "{\n" +
"      \"query\": {\n" +
"        \"term\": {\n" +
"          \"appli_name\": \"{{name}}\"\n" +
"        }\n" +
"      }\n" +
"}";
elasticsearchTemplate.searchTemplateBySource(param,templatesource,Main2.class).f
orEach(s -> System.out.println(s));
```

搜索建议Completion Suggester

搜索建议功能能够快速提示要搜索的内容（请参考百度搜索功能），搜索建议字段需要配置suggest属性为true

```
/**
 * 搜索建议方法
 * @param fieldName 要搜索的字段（需要设置该字段mapping配置suggest属性为true）
 * @param fieldValue 要搜索的内容
 * @param clazz
 * @return 返回搜索结果列表 搜索建议默认条数为10条
 * @throws Exception
 */
public List<String> completionSuggest(String fieldName,String
fieldValue,Class<T> clazz) throws Exception;
```

```
List<String> list = elasticsearchTemplate.completionSuggest("appli_name", "1",
Main2.class);
list.forEach(main2 -> System.out.println(main2));
```

```
@ESMapping(suggest = true)
private String appli_name;
```

搜索建议phrase suggest

```
/**
* 搜索建议Phrace Suggester
* @param fieldName
* @param fieldValue
* @param param 定制Phrace Suggester的参数
* @param clazz
* @return
* @throws Exception
*/
public List<String> phraseSuggest(String fieldName, String fieldValue,
ElasticsearchTemplateImpl.PhraseSuggestParam param, Class<T> clazz) throws
Exception;
```

```
ElasticsearchTemplateImpl.PhraseSuggestParam param = new
ElasticsearchTemplateImpl.PhraseSuggestParam(5,1,null,"always");
elasticsearchTemplate.phraseSuggest("body", "who is good boy zhangxinpen may be
a goop",param, Sugg.class).forEach(s -> System.out.println(s));
//可以使用默认的参数，将param传入null
elasticsearchTemplate.phraseSuggest("body", "who is good boy zhangxinpen may be
a goop",null, Sugg.class).forEach(s -> System.out.println(s));
```

参数如何定义详见官方文档:
https://www.elastic.co/guide/en/elasticsearch/reference/7.3/search-suggesters.html

根据ID查询

```
/**
 * 根据ID查询
 * @param id 主键值，对应第二泛型类型
 * @param clazz
 * @return
 * @throws Exception
 */
public T getById(M id, Class<T> clazz) throws Exception;
```

```
Main2 main2 = elasticsearchTemplate.getById("main2", Main2.class);
System.out.println(main2);
```

mget查询

```
/**
 * 根据ID列表批量查询
 * @param ids ID主键数组
 * @param clazz
 * @return
 * @throws Exception
 */
public List<T> mgetById(M[] ids, Class<T> clazz) throws Exception;
```

```
String[] list = {"main2","main3"};
List<Main2> listResult = elasticsearchTemplate.mgetById(list, Main2.class);
listResult.forEach(main -> System.out.println(main));
```

QueryBuilder常用用法展示

精准查询

```
//精准查询的字段需要设置keyword属性（默认该属性为true），查询时fieldname需要带上.keyword
QueryBuilder queryBuilder = QueryBuilders.termQuery("appli_name.keyword","456");
List<Main2> list = elasticsearchTemplate.search(queryBuilder,Main2.class);
list.forEach(main2 -> System.out.println(main2));
//如果field类型直接为keyword可以不用加.keyword
```

短语查询

```
//中国好男儿
//必须相邻的查询条件
QueryBuilder queryBuilder = QueryBuilders.matchPhraseQuery("appli_name","国好");
```

相关度查询

```
//中国好男儿
//slop设置为2，中和男最多能移动两次并完成匹配
QueryBuilder queryBuilder = QueryBuilders.matchPhraseQuery("appli_name","中
男").slop(2);
```

范围查询

```
QueryBuilder queryBuilder =
QueryBuilders.rangeQuery("sum_premium").from(1).to(3);
```

全文匹配

```
QueryBuilder queryBuilder = QueryBuilders.matchQuery("appli_name","中男儿");
```

minimumShouldMatch最少匹配参数

```
//"中 男 儿 美 丽 人 生"最少匹配词语数量是75%，该查询查不到信息
QueryBuilder queryBuilder = QueryBuilders.matchQuery("appli_name","中 男 儿 美 丽 人 生").minimumShouldMatch("75%");
```

match查询集成fuzzy纠错查询

```
QueryBuilder queryBuilder = QueryBuilders.matchQuery("appli_name","spting");
((MatchQueryBuilder) queryBuilder).fuzziness(Fuzziness.AUTO);
```

match查询设定查询条件逻辑关系

```
//默认是OR
QueryBuilder queryBuilder = QueryBuilders.matchQuery("appli_name","spring sps").operator(Operator.AND);
```

fuzzy纠错查询

```
//原文是spring，查询条件输入为spting也能查询到结果
QueryBuilder queryBuilder = QueryBuilders.fuzzyQuery("appli_name","spting");
```

boost权重设置

```
//查询结果appli_name为spring的会被优先展示其次456，再次123
QueryBuilder queryBuilder1 =
QueryBuilders.termQuery("appli_name.keyword","spring").boost(5);
QueryBuilder queryBuilder2 =
QueryBuilders.termQuery("appli_name.keyword","456").boost(3);
QueryBuilder queryBuilder3 =
QueryBuilders.termQuery("appli_name.keyword","123");
BoolQueryBuilder queryBuilder = QueryBuilders.boolQuery();
queryBuilder.should(queryBuilder1).should(queryBuilder2).should(queryBuilder3);
```

prefix前缀查询

```
//性能差，扫描整个倒排索引，前缀越短，要处理的doc越多，性能越差，尽可能用长前缀搜索
//查询appli_name字段值前缀为1的内容
QueryBuilder queryBuilder = QueryBuilders.prefixQuery("appli_name","1");
```

wildcard通配符查询

```
//性能较差不建议使用
//?：任意字符
//*：0个或任意多个字符
QueryBuilder queryBuilder = QueryBuilders.wildcardQuery("appli_name","1?3");
```

regexp正则查询

```
//性能较差不建议使用
QueryBuilder queryBuilder = QueryBuilders.regexpQuery("appli_name","[0-9].+");
//[0-9]：指定范围内的数字
//[a-z]：指定范围内的字母
//.：一个字符
//+：前面的正则表达式可以出现一次或多次
```

组合逻辑查询

组合逻辑查询是多种查询方式的逻辑组合，共有三种：与must、或should、非mustNot

```
//select * from Main2 where (appli_name = 'spring' or appli_name = '456') and
risk_code = '0101' and proposal_no != '1234567'
QueryBuilder queryBuilder1 =
QueryBuilders.termQuery("appli_name.keyword","spring");
QueryBuilder queryBuilder2 =
QueryBuilders.termQuery("appli_name.keyword","456");
QueryBuilder queryBuilder3 = QueryBuilders.termQuery("risk_code","0101");
QueryBuilder queryBuilder4 =
QueryBuilders.termQuery("proposal_no.keyword","1234567");
BoolQueryBuilder queryBuilder = QueryBuilders.boolQuery();
queryBuilder.should(queryBuilder1).should(queryBuilder2);
queryBuilder.must(queryBuilder3);
queryBuilder.mustNot(queryBuilder4);
```

Dis Max Query

所有查询条件中只取匹配度最高的那个条件的分数作为最终分数

```
//假设
//第一条数据title匹配bryant fox的分数为0.8 body匹配bryant fox的分数为0.1，这条数据最终得
分为0.8
//第二条数据title匹配bryant fox的分数为0.6 body匹配bryant fox的分数为0.7，这条数据最终得
分为0.7
//dis_max查询后第一条数据相关度评分更高，排在第二条数据的前面
//如果不用dis_max，则第二条数据的得分为1.4高于第一条数据的0.9
//如果再附加其他匹配结果的分数，需要指定tieBreaker
//获得最佳匹配语句句的评分_score
//将其他匹配语句句的评分与tie_breaker 相乘
//对以上评分求和并规范化
//Tier Breaker：介于0-1 之间的浮点数（0代表使用用最佳匹配；1 代表所有语句句同等重要）
QueryBuilders.disMaxQuery()
.add(QueryBuilders.matchQuery("title", "bryant fox"))
.add(QueryBuilders.matchQuery("body", "bryant fox"))
.tieBreaker(0.2f);
```

Multi Match Query

最佳匹配best_fields：和Dis Max Query效果一样

```
QueryBuilders.multiMatchQuery("Quick pets", "title","body")
.minimumShouldMatch("20%")
.type(MultiMatchQueryBuilder.Type.BEST_FIELDS)
.tieBreaker(0.2f);
```

最多匹配most_fields：能匹配到更多字段的记录优先（不管其中某一个字段有多么匹配）

```
PUT /mult/_doc/1
{
    "s1": "shanxi shanxi shanxi shanxi shanxi",
    "s2": "shanxi",
    "s3": "ttttt",
    "s4": "Brown rabbits are commonly seen."
```

```
}
PUT /mult/_doc/2
{
    "s1": "datong",
    "s2": "datong",
    "s3": "datong",
    "s4": "Brown rabbits are commonly seen."
}
//虽然shanxi在s1中出现了很多次，但是只出现了两个字段s1 s2
//datong出现在了三个字段s123，所以优先更多出现字段的这条记录
//doc2的分数大于doc1的分数
QueryBuilders.multiMatchQuery("shanxi datong", "s1","s2","s3","s4")
.type(MultiMatchQueryBuilder.Type.MOST_FIELDS);
```

跨字段匹配cross_fields：综合起来最匹配的，即类似将所有字段合并到一个字段中，搜索这个字段看谁分高
最佳实践：可以代替copy_to节省了一个字段的倒排空间

```
PUT /mult/_doc/3
{
    "s1": "sichuan sichuan",
    "s2": "sichuan",
    "s3": "eee",
    "s4": "sichuan"
}

PUT /mult/_doc/4
{
    "s1": "chengdu t chengdu chengdu",
    "s2": "rr",
    "s3": "ss",
    "s4": "Brown rabbits are commonly seen."
}
//虽然chengdu在s1中出现了3次，但是合在一起显然doc3更匹配（TF更高、文档长度小，IDF一致）
//所以doc3分数更高
QueryBuilders.multiMatchQuery("chengdu sichuan", "s1","s2","s3","s4")
.type(MultiMatchQueryBuilder.Type.CROSS_FIELDS);
```

Function Score Query

https://www.elastic.co/guide/en/elasticsearch/reference/7.1/query-dsl-function-score-query.html
可以在查询结束后，对每一个匹配的文档进行一系列的重新算分，根据新生成的分数进行排序

```
//新的算分 = 老的算分 * log( 1 + factor*votes的值)
//相当于和数据中的内容进行加权，不是直接指定加权值，而是指定加权策略，数据中的字段可以直接影响到算分
ScoreFunctionBuilder<?> scoreFunctionBuilder = ScoreFunctionBuilders
.fieldValueFactorFunction("votes")
.modifier(FieldValueFactorFunction.Modifier.LOG1P)
.factor(0.1f);
QueryBuilders.functionScoreQuery(QueryBuilders.matchQuery("title", "bryant fox"),scoreFunctionBuilder)
.boostMode(CombineFunction.MULTIPLY)//默认就是乘
.maxBoost(3f);
```

boosting Query

https://www.elastic.co/guide/en/elasticsearch/reference/7.1/query-dsl-boosting-query.html
boosting查询是定制一组查询策略的方式，它可以定制一个否定查询条件组，并设定这个条件组降低匹配的程度
positive：这个查询返回的信息必须匹配（boostingQuery的第一个参数）
negative：这个可以根据negative_boost来决策对搜索结果相关度的调整（boostingQuery的第二个参数）
negative_boost：参数boost的含义
- 当boost > 1 时，打分的相关度相对性提升
- 当0 < boost < 1 时，打分的权重相对性降低

```
QueryBuilders.boostingQuery(QueryBuilders.matchQuery("title", "bryant fox"),
QueryBuilders.matchQuery("flag", "123")).negativeBoost(0.2f);
```

注意与boost进行区分，boosting Query有一套固定的策略

过滤器

> 过滤器查询需要和布尔查询结合使用，效果上和普通查询没有什么区别

> 比如下面的例子must和filter的关系是并且

> 但过滤器查询首先不会计算相关度评分，而普通查询会去计算相关度评分，并且按照相关对进行排序，那么如果你并不需要相关度评分就要优先选择过滤器查询

> 另外过滤器查询内置了bitset的caching机制，能够非常大程度的提升查询的性能，减少扫描倒排索引的几率，所以在日常开发中，我们需要认真评估，积极使用过滤器查询

```
//select * from Main2 where appli_name = '456' and risk_code = '0101'
QueryBuilder queryBuilder =
QueryBuilders.boolQuery().must(QueryBuilders.termQuery("appli_name.keyword","456
"))
//下面如果会比较频繁的作为子集合，就比较适合通过filter来缓存
.filter(QueryBuilders.matchPhraseQuery("risk_code","0101"));
List<Main2> list = elasticsearchTemplate.search(queryBuilder,Main2.class);
list.forEach(main2 -> System.out.println(main2));
```

> 更多QueryBuilder详见https://www.elastic.co/guide/en/elasticsearch/client/java-rest/6.6/java-rest-high-query-builders.html

nested查询

```
NestedQueryBuilder queryBuilder =
QueryBuilders.nestedQuery("actors",QueryBuilders.matchQuery("actors.first_name",
"DiCaprio1"), ScoreMode.Total);
elasticsearchTemplate.search(queryBuilder, MyMovies.class).forEach(s ->
{System.out.println(s);});
```

NestedQueryBuilder需要传入一个nestedPath

```
NestedQueryBuilder queryBuilder =
QueryBuilders.nestedQuery("nestedPath",QueryBuilders.termsQuery("nestedPath.id",
1),ScoreMode.Total);
```

按照多索引查询说明

有两种方式可供多索引查询

1. 通过配置注解`searchIndexNames`，这种方式可以在默认能查询多索引的所有api中生效，如果配置此项，再相应的查询方法将会查询多个索引，并按照当前poji的字段结果进行返回，但由于通过注解配置不灵活，所以如果不是特别确定的场景并不建议这么做。

```
@ESMetaData(indexName = "main5",searchIndexNames = {"main5","index"},
number_of_shards = 5,number_of_replicas = 0,printLog = false)
public class Main5 implements Serializable {

//查询api调用不发生任何变化
```

2. 通过api传入需要查询的多个索引名称，这种方式相比注解方式更加灵活可靠，如果涉及跨索引查询的业务推荐使用这种方法，目前已经添加了普通查询对应的多索引入参api，后续将添加聚合查询的跨索引查询

```
//传入main5、main6作为需要被2个索引范围
List<Main6> list =
elasticsearchTemplate.search(QueryBuilders.matchAllQuery(),Main6.class,"main5","
main6");
System.out.println(list.size());
//查询结果仅包含main6的字段结果
list.forEach(main6 -> System.out.println(main6));
```

这是一个新增的接口方法实例，均在最后添加了可变长入参的方式

```
 /**
     * 非分页查询(跨索引)
     * 目前暂时传入类类型
     * @param queryBuilder
     * @param clazz
     * @return
     * @throws Exception
     */
    public List<T> search(QueryBuilder queryBuilder, Class<T> clazz,String...
indexs) throws Exception;
```

==建议跨索引查询时多索引之间尽量字段重合度高==

聚合查询

原生聚合查询

```
/**
 * 聚合基础方法，此方法较少封装，其目的为了满足工具无法实现的聚合查询
 * @param aggregationBuilder 聚合内容
 * @param queryBuilder 检索条件
 * @param clazz
 * @return 聚合结果（官方原生聚合结果）
 * @throws Exception
 */
public Aggregations aggs(AggregationBuilder aggregationBuilder, QueryBuilder
queryBuilder, Class<T> clazz) throws Exception;
```

```
SumAggregationBuilder aggregation =
AggregationBuilders.sum("agg").field("sum_amount");
Aggregations aggregations =
elasticsearchTemplate.aggs(aggregation,null,Main2.class);
Sum agg = aggregations.get("agg");
double value = agg.getValue();
System.out.println(value);
```

> 更详细用法请参考官方文档https://www.elastic.co/guide/en/elasticsearch/client/java-api/6.6/_metrics_aggregations.html

> 官方所有聚合Builder请参考https://www.elastic.co/guide/en/elasticsearch/client/java-rest/6.6/java-rest-high-aggregation-builders.html

普通聚合查询

```
/**
 * 以aggstypes的方式metric度量，该方法不做分组，直接聚合统计metricName字段的sum、count、
avg、min、max
 * @param metricName 需要统计分析的字段
 * @param aggsType sum、count、avg、min、max类型
 * @param queryBuilder 查询条件，如果不需要传入null即可
 * @param clazz
 * @return 返回sum、count、avg、min、max的结果
 * @throws Exception
 */
public double aggs(String metricName, AggsType aggsType, QueryBuilder
queryBuilder, Class<T> clazz) throws Exception;
```

```
double sum = elasticsearchTemplate.aggs("sum_premium",
AggsType.sum,null,Main2.class);
double count = elasticsearchTemplate.aggs("sum_premium",
AggsType.count,null,Main2.class);
double avg = elasticsearchTemplate.aggs("sum_premium",
AggsType.avg,null,Main2.class);
double min = elasticsearchTemplate.aggs("sum_premium",
AggsType.min,null,Main2.class);
//如果翻译成sql：select max(sum_premium) from Main2
double max = elasticsearchTemplate.aggs("sum_premium",
AggsType.max,null,Main2.class);


System.out.println("sum====="+sum);
System.out.println("count====="+count);
System.out.println("avg====="+avg);
System.out.println("min====="+min);
System.out.println("max====="+max);
```

分组普通聚合查询

```
/**
 * 普通聚合查询，此方法最常用
 * 以bucket分组以aggstypes的方式metric度量
 * @param bucketName 以bucketName字段进行分组，在es中这个是“桶”的概念
 * @param metricName 需要统计分析的字段
 * @param aggsType
 * @param clazz
 * @return
 */
public Map aggs(String metricName, AggsType aggsType,QueryBuilder queryBuilder,
Class<T> clazz, String bucketName) throws Exception;
```

```
//如果翻译成sql：select appli_name,max(sum_premium) from Main2 group by appli_name
Map map = elasticsearchTemplate.aggs("sum_premium",
AggsType.sum,null,Main2.class,"appli_name");
map.forEach((k,v) -> System.out.println(k+"     "+v));
```

==默认按照聚合结果降序排序==

下钻（2层）聚合查询

```
/**
 * 下钻聚合查询(无排序默认策略)，2次分组，注意目前此方法仅支持2层分组
 * 以bucket分组以aggstypes的方式metric度量
 * @param metricName 需要统计分析的字段
 * @param aggsType
 * @param queryBuilder
 * @param clazz
 * @param bucketNames 下钻分组字段
 * @return
 * @throws Exception
 */
public List<Down> aggswith2level(String metricName, AggsType
aggsType,QueryBuilder queryBuilder, Class<T> clazz ,String... bucketNames)
throws Exception;
```

```
//select appli_name,risk_code,sum(sumpremium) from Main2 group by
appli_name,risk_code
List<Down> list = elasticsearchTemplate.aggswith2level("sum_premium",
AggsType.sum,null,Main2.class,"appli_name","risk_code");
list.forEach(down ->
    {
        System.out.println("1:"+down.getLevel_1_key());
        System.out.println("2:"+down.getLevel_2_key() + "    "+
down.getValue());
    }
);
```

统计聚合查询

```
/**
 * 统计聚合metric度量
 * @param metricName 需要统计分析的字段
 * @param queryBuilder
 * @param clazz
 * @return
 * @throws Exception
 */
public Stats statsAggs(String metricName, QueryBuilder queryBuilder, Class<T>
clazz) throws Exception;
```

```
//此方法可以一次查询便返回针对metricName统计分析的sum、count、avg、min、max指标值
Stats stats = elasticsearchTemplate.statsAggs("sum_premium",null,Main2.class);
System.out.println("max:"+stats.getMax());
System.out.println("min:"+stats.getMin());
System.out.println("sum:"+stats.getSum());
System.out.println("count:"+stats.getCount());
System.out.println("avg:"+stats.getAvg());
```

分组统计聚合查询

```
/**
 * 以bucket分组，统计聚合metric度量
 * @param bucketName 以bucketName字段进行分组
 * @param metricName 需要统计分析的字段
 * @param queryBuilder
 * @param clazz
 * @return
 * @throws Exception
 */
public Map<String,Stats> statsAggs(String metricName, QueryBuilder queryBuilder,
Class<T> clazz, String bucketName) throws Exception;
```

和上一个方法相比，这个方法增加了分组的功能

```
Map<String,Stats> stats =
elasticsearchTemplate.statsAggs("sum_premium",null,Main2.class,"risk_code");
stats.forEach((k,v) ->
    {
        System.out.println(k+"    count:"+v.getCount()+"
sum:"+v.getSum()+"...");
    }
);
```

基数查询

```
/**
 * 基数查询，即count(distinct)返回一个近似值，并不一定会非常准确
 * @param metricName 需要统计分析的字段
 * @param queryBuilder
 * @param clazz
 * @return
 * @throws Exception
 */
public long cardinality(String metricName, QueryBuilder queryBuilder, Class<T>
clazz) throws Exception;
```

```
//select count(distinct proposal_no) from Main2
long value = elasticsearchTemplate.cardinality("proposal_no",null,Main2.class);
System.out.println(value);
```

百分比聚合查询

```
/**
 * 百分比聚合 默认按照50%,95%,99%（TP50 TP95 TP99）进行聚合
 * @param metricName 需要统计分析的字段,需要是数字类型
 * @param queryBuilder
 * @param clazz
 * @return
 * @throws Exception
 */
public Map percentilesAggs(String metricName, QueryBuilder queryBuilder,
Class<T> clazz) throws Exception;

/**
 * 以百分比聚合 自定义百分比段位
```

```
 * @param metricName 需要统计分析的字段,需要是数字类型
 * @param queryBuilder
 * @param clazz
 * @param customSegment 自定义的百分比段位
 * @return
 * @throws Exception
 */
public Map percentilesAggs(String metricName, QueryBuilder queryBuilder,
Class<T> clazz,double... customSegment) throws Exception;
```

百分比聚合即查询统计字段50%的数据在什么值以内，95%的数据在什么值以内

```
//下面的例子是取sum_premium这个字段的TP50 TP95 TP99
Map map = elasticsearchTemplate.percentilesAggs("sum_premium",null,Main2.class);
map.forEach((k,v) ->
        {
            System.out.println(k+"     "+v);
        }
);
//输出结果是：
50.0     3.5
95.0     6.0
99.0     6.0
//即50%的sum_premium在3.5以下
//即95%的sum_premium在6.0以下
//即99%的sum_premium在6.0以下

//也可以自定义百分比段位
Map map =
elasticsearchTemplate.percentilesAggs("sum_premium",null,Main2.class,10,20,30,50
,60,90,99);
```

百分等级聚合查询

```
/**
 * 以百分等级聚合 (统计在多少数值之内占比多少)
 * @param metricName 需要统计分析的字段,需要是数字类型
 * @param queryBuilder
 * @param clazz
 * @param customSegment
 * @return
 * @throws Exception
 */
public Map percentileRanksAggs(String metricName, QueryBuilder queryBuilder,
Class<T> clazz,double... customSegment) throws Exception;
```

百分等级聚合即给定一个统计结果的段位，并查询在段位范围内出现的百分比是多少

```
//我们给定一个sum_premium字段的统计段位1,4,5,9，即1以下、4以下、5以下、9以下
//最终获取在上述范围内数据出现的比百分
Map map =
elasticsearchTemplate.percentileRanksAggs("sum_premium",null,Main2.class,1,4,5,9
);
map.forEach((k,v) ->
        {
            System.out.println(k+"     "+v);
```

```
        }
);

//输出结果为：
8.333333333333332      1.0
58.333333333333336     4.0
75.0                   5.0
100.0                  9.0
//即8.3%的数据sum_premium字段值在1以下
//即58.3%的数据sum_premium字段值在4以下
//即75%的数据sum_premium字段值在5以下
//即100%的数据sum_premium字段值在9以下
```

过滤器聚合查询

```
/**
 * 过滤器聚合，可以“变”的分组
 * @param metricName 需要统计分析的字段
 * @param aggsType 统计类型
 * @param clazz
 * @param queryBuilder
 * @param filters FiltersAggregator过滤器封装对象，每个过滤器查询出的数据结果都可以作为一个桶
 * @return
 * @throws Exception
 */
public Map filterAggs(String metricName, AggsType aggsType, QueryBuilder queryBuilder,Class<T> clazz, FiltersAggregator.KeyedFilter... filters) throws Exception;
```

过滤器聚合让es的聚合功能非常的灵活，它可以灵活定制分组规格

```
//以下的例子是以risk_code是0101为1组，risk_code是0103或0104为1组，求sum_premium在分组内的和
Map map = elasticsearchTemplate.filterAggs("sum_premium", AggsType.sum, null,Main2.class,
        new FiltersAggregator.KeyedFilter("0101", QueryBuilders.matchPhraseQuery("risk_code", "0101")),
        new FiltersAggregator.KeyedFilter("0103或104", QueryBuilders.matchQuery("risk_code", "0103 0104")));
map.forEach((k, v) ->
    System.out.println(k + "    " + v)
);
```

直方图聚合查询

```
/**
 * 直方图聚合
 * @param metricName 需要统计分析的字段
 * @param aggsType 统计类型
 * @param queryBuilder
 * @param clazz
 * @param bucketName 以bucketName字段进行分组
 * @param interval 分组字段值的间隔
 * @return
 * @throws Exception
 */
public Map histogramAggs(String metricName,  AggsType aggsType,QueryBuilder
queryBuilder,Class<T> clazz,String bucketName,double interval) throws Exception;
```

直方图聚合是统计针对分组字段，每隔X的值统计一次度量值

```
//统计sum_premium的数值每3的倍数，统计一次proposal_no的个数
Map map = elasticsearchTemplate.histogramAggs("proposal_no", AggsType.count,
null,Main2.class,"sum_premium",3);
map.forEach((k, v) ->
        System.out.println(k + "    " + v)
);
//输出结果为：
0.0    2
3.0    3
6.0    1
```

日期直方图聚合查询

```
/**
 * 日期直方图聚合
 * @param metricName 需要统计分析的字段
 * @param aggsType 统计类型
 * @param queryBuilder
 * @param clazz
 * @param bucketName
 * @param interval 日期分组间隔
 * @return
 * @throws Exception
 */
public Map dateHistogramAggs(String metricName, AggsType aggsType, QueryBuilder
queryBuilder, Class<T> clazz, String bucketName, DateHistogramInterval interval)
throws Exception;
```

日期直方图与直方图类似，只是将分组的字段替换为日期类型

```
//统计input_date每两个小时sum_premium的金额总合
Map map = elasticsearchTemplate.dateHistogramAggs("sum_premium", AggsType.sum,
null,Main2.class,"input_date", DateHistogramInterval.hours(2));
map.forEach((k, v) ->
        System.out.println(k + "    " + v)
);
```

更多聚合查询的方式

es支持的聚合方式远不止于此，工具只是针对最常用的一部分查询方式进行封装，以减轻代码量

例如我们需要实现一个范围聚合，可以通过以下例子实现：

```
//以sum_premium的范围分组，并统计sum_premium的数量
AggregationBuilder aggregation =
AggregationBuilders.range("range").field("sum_premium").addUnboundedTo(1).addRan
ge(1,4).addRange(4,100).addUnboundedFrom(100);
aggregation.subAggregation(AggregationBuilders.count("agg").field("proposal_no.k
eyword"));
Aggregations aggregations =
elasticsearchTemplate.aggs(aggregation,null,Main2.class);
Range range = aggregations.get("range");
for (Range.Bucket entry : range.getBuckets()) {
    ValueCount count = entry.getAggregations().get("agg");
    long value = count.getValue();
    System.out.println(entry.getKey() + "    " + value);
}
```

> 更多聚合API用法详见https://www.elastic.co/guide/en/elasticsearch/client/java-rest/6.6/java-rest-high-aggregation-builders.html

运维功能

打印请求es服务日志

如果需要调试，需要获取请求es的json报文，可以配置es索引结构实体类注解的打印报文开关 `printLog = true`

```
@ESMetaData(indexName = "index", number_of_shards = 5,number_of_replicas =
0,printLog = true)
public class Main2 implements Serializable {
```

此项配置默认为关闭，开启后仅仅支持几个常用功能的日志打印，如果需要支持更多的功能打印，请在相应位置添加如下代码即可

```
if(metaData.isPrintLog()){
    logger.info(searchSourceBuilder.toString());
}
```

打印自动注册的情况

如果需要确认自动注册的情况（包括：需要自动创建索引结构的信息以及自动生成ESCRepository代理类的信息）可以配置EnableESTools注解printregmsg属性为true

```
@EnableESTools(entityPath = {printregmsg = true)
```

测试demo包（附件testdemo.zip）说明

请构建一个springboot程序，并引入esclientrhl，配置好es服务即可做相关测试demo的调用

- TestAggs是测试聚合相关的方法
- TestCRUD是测试索引数据增删改查的相关方法

- TestIndex是测试创建删除索引的相关方法
- TestLowLevelClient是测试LowLevelClient的方法
  方法上我没写注释，请大家对照readme