EsClientRHL

- 使用前你应该具有哪些技能
- 工具功能范围介绍
    - 索引结构管理
    - CRUD
    - 聚合查询
- 工具源码结构介绍
    - annotation
    - config
    - enums
    - index
    - repository
    - util
- 开始使用
    - 使用前说明
    - maven依赖
    - 引入组件
    - 使用组件
    - 索引管理功能
        - 元数据配置
        - 索引结构配置
        - 根据配置信息自动创建索引结构mapping
        - 手工创建或删除索引结构
        - 判断索引是否存在
    - CRUD功能说明
        - LowLevelClient查询
        - 新增索引数据
        - 批量新增索引数据
        - 部分更新索引数据
        - 覆盖更新索引数据
        - 删除索引数据
        - 根据查询条件删除索引数据
        - 判断索引数据是否存在
        - 原生查询
        - 支持、查询条件的定制查询
        - 支持分页、高亮、排序、查询条件的定制查询
        - count查询
        - scroll查询
        - ~~模版查询~~
        - 搜索建议
        - 根据ID查询
        - mget查询
    - QueryBuilder常用用法展示
        - 精准查询
        - 短语查询
        - 相关度查询
        - 范围查询

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。

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
- 覆盖更新索引数据
- 部分更新索引数据
- 删除索引数据
- 判断索引数据是否存在
- 原生查询

- 支持、查询条件的定制查询
- 支持分页、高亮、排序、查询条件的定制查询
- count查询
- scroll查询（用于大数据量查询）
- ~~模版查询~~
- 搜索建议
- 根据ID查询
- mget查询
- 按照多索引查询说明（2013-03-19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

maven依赖

请把组件安装到maven仓库

```
<dependency>
    <groupId>org.zxp</groupId>
    <artifactId>esclientrhl</artifactId>
    <version>1.0.0</version>
</dependency>
```

也可以直接引入starter

请下载esclientrhl-springboot-starter

```
<dependency>
    <groupId>org.zxp</groupId>
    <artifactId>esclientrhl-springboot-starter</artifactId>
    <version>1.0.0</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>6.6.0</elasticsearch.version>
……
</properties>
```

引入组件

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

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

如果引入的是esclientrhl-starter，则启动类上无需添加@EnableESTools，会自动扫描启动类路径下的包，除非有额外的包需要配置，否则无需配置@EnableESTools

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

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

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;
```

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",indexType = "main4", 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，必须配置，墙裂建议每个index下只有一个type
 */
String indexType();
/**
 * 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autocomplete，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 autocomplete() 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 "";
```

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")
```

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",indexType = "main", 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);
```

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);
```

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

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);
```

根据查询条件删除索引数据

```
elasticsearchTemplate.deleteByCondition(QueryBuilders.matchQuery("appl
i_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、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));
```

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

```
/**
 * 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
 */
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
 */
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

暂时无法使用该方法，原因为官方API SearchTemplateRequestBuilder仍保留对transportClient 的依赖，但Migration Guide 中描述需要把transportClient迁移为RestHighLevelClient

搜索建议

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;
```

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(queryB
uilder3);
```

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);
```

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.key
word","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*

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",indexType = "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_c
ode");
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,1
0,20,30,50,60,90,99);
```

- 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.cla
ss,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).addRange(1,4).addRange(4,100).addUnboundedFrom(100);
aggregation.subAggregation(AggregationBuilders.count("agg").field("pro
posal_no.keyword"));
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",indexType = "main4", 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());
}
```

测试demo包（根目录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