Elasticsearch

标签　　空

新增属性

Elasticsearch 是一个分布式、可扩展、实时的**搜索与数据分析**引擎。它将下面功能打包成一个单独的服务，并提供的简单的 RESTful API 进行通信：

- 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

教程地址：https://www.elastic.co/guide/cn/elasticsearch/guide/current/getting-started.html

面向文档介绍

面向文档：在表格式数据存储时需要将对象转为行列，在查询时又需要转为对象，使用面向文档可直接免除这些步骤来存储对象信息。es 中不仅可直接存储 JSON 格式的文档，还能使文档能够被检索。

存储一个 User 对象的示例：

JSON

```
{
    "email":      "john@smith.com",
    "first_name": "John",
    "last_name":  "Smith",
    "info": {
        "bio":         "Eco-warrior and defender of the weak",
        "age":         25,
        "interests": [ "dolphins", "whales" ]
    },
    "join_date": "2014/05/01"
}
```

一个 ES 集群可包含多个索引（库），一个索引可包含多个类型（表,但不用专门创建），一个类型可包含多个文档（行），一个文档可包含多个属性（列）。

Server 准备

搭建 server

官网下载地址：https://www.elastic.co/cn/downloads/elasticsearch

启动：windows 上运行 bin\elasticsearch.bat，linux 使用下面命令

Bash

```
# 进入安装目录
cd elasticsearch-<version>

# 启动
```

```
./bin/elasticsearch
# 后台启动
./bin/elasticsearch -d
```

我们可用有下面两种方式来与 es 交互：

- Java API：通过代码内置的客户端与 es 交互，分为节点客户端、传输客户端，均使用 9300 端口
- RestFull API：通过客户端发送 http 请求与 es 交互，使用 9200 端口

可视化工具

Client 操作

搭建 Client

JAVA 代码: RestHighLevelClient

ES 依赖

XML

```xml
<dependency><!-- es客户端rest api -->
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
    <version>6.2.4</version>
</dependency>

<dependency><!-- orm  -->
    <groupId>com.bbossgroups.plugins</groupId>
    <artifactId>bboss-elasticsearch-rest-jdbc</artifactId>
    <version>6.6.0</version>
</dependency>
<dependency>
    <groupId>com.bbossgroups.plugins</groupId>
    <artifactId>bboss-elasticsearch-spring-boot-starter</artifactId>
    <version>6.6.0</version>
</dependency>
```

ES 配置

YAML

```yaml
spring:
  elasticsearch:
    bboss:
      elasticsearch:
        rest:
          hostNames: localhost:9200
      elasticUser: root
      elasticPassword: root
```

创建客户端代码

Java

```java
@Configuration
public class ElasticsearchConfig {

    @Value("${spring.elasticsearch.bboss.elasticsearch.rest.hostNames}")
    private String path;

    @Value("${spring.elasticsearch.bboss.elasticUser}")
    private String username;

    @Value("${spring.elasticsearch.bboss.elasticPassword}")
    private String password;


    @Bean
    public RestHighLevelClient restHighLevelClient(){
        // 回调函数
        RestClientBuilder.HttpClientConfigCallback callback = clientBuilder -> {
            // 创建凭证
            final CredentialsProvider provider = new BasicCredentialsProvider();
```

```
            provider.setCredentials(AuthScope.ANY,new UsernamePasswordCredentials(username, password));
            clientBuilder.disableAuthCaching();
            return clientBuilder.setDefaultCredentialsProvider(provider);
        };

        // 创建客户端
        RestHighLevelClient client = new RestHighLevelClient(
                RestClient.builder(
                        new HttpHost("192.168.35.100", 9200, "http"),
                        new HttpHost("192.168.35.200", 9200, "http")
                ).setHttpClientConfigCallback(callback)
        );

        return client;
    }
    // Getter\Setter
}
```

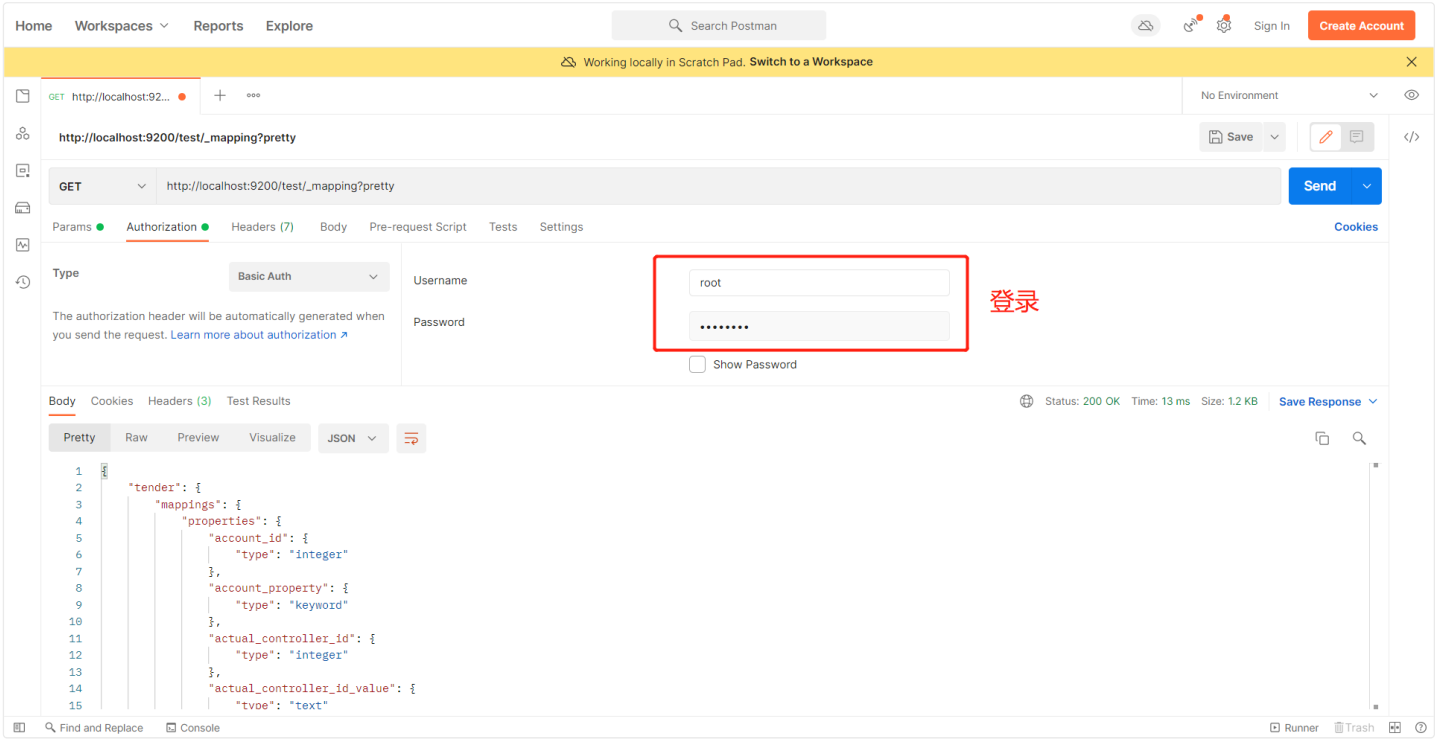
使用客户端

Java

```
@Autowired
private RestHighLevelClient client;
```

REST 请求

直接使用 http 请求工具访问即可，postman 访问带密码的 es server 如下：



索引管理 | index

JAVA 代码: RestHighLevelClient

Java

REST 请求

查询全部索引：GET

Ada

```
-- 配置信息：健康状态、打开状态、名称、uuid、主分片数、副分片数、文档数量、是否删除、总分片大小、主分片大小
http://localhost:9200/_cat/indices
```

查询单个索引：GET

```
Ada
http://localhost:9200/my_index
```

创建索引：PUT

```
Ada
http://localhost:9200/my_index
{

    -- 映射信息(可选)
    "mappings": {
        -- 创建类型user
        "user": {
            "properties": {
                "age": { "type": "long" },
            }
        }
    }
    -- 索引配置(可选)
    "settings":{
        "number_of_shards":1,    //设置分片数量
        "number_of_replicas":2,  //设置副本数量
        "index":{                //自定义索引默认分析器
          "analysis":{
             "analyzer":{
                "default":{
                   "tokenizer":"standard",                                    //分词器
                   "filter":[ "asciifolding", "lowercase", "ourEnglishFilter" ] //过滤器
                }
             },
             "filter":{
                "ourEnglishFilter":{ "type":"kstem" }
             }
          }
        }
    }
}
```

删除索引：DELETE

```
Ada
http://localhost:9200/my_index
```

类型映射 | mapping

类型是在添加映射时被创建的(也可以不指定，即索引本身仅存一个类型)，类型映射指定文档属性的描述信息，类似 mysql 中的字段信息，es 中映射类型有如下几种：

- 字符串: `string`
- 整数 : `byte`, `short`, `integer`, `long`
- 浮点数: `float`, `double`
- 布尔型: `boolean`
- 日期: `date`

注意：映射字段被设置后无法删除且无法修改已设置的映射属性（像修改只能查询复制→删除索引→粘贴新建）。

JAVA 代码: RestHighLevelClient

```
Java
```

REST 请求

查询全部映射：GET

```
Ada
http://localhost:9200/my_index/my_type/_mapping?pretty
```

查询单个映射：GET

```
Ada
```

```
http://localhost:9200/my_index/my_type/_mapping/field/username
```

新建映射：PUT

```
-- 映射信息也可以在创建索引时指定，如果映射字段已存在则无法添加，如果不存在则会补上(即只能补充，不能删改)
http://localhost:9200/my_index/my_type/_mapping?pretty
{
    "properties":{
      "my_name1":{"type" : "integer"},
      "my_name2": { "type": "date" },
      -- 级联类型映射
      "my_type2":{
          "properties" : { ... }
      }
    }
}
```

文档管理 | doc

JAVA代码: RestHighLevelClient

```
Java
```

REST请求

查询文档列表：GET

```
Ada
```

查询单个文档：GET

```
http://localhost:9200/my_index/my_type/1
```

新建文档：PUT

```
-- 索引为testdb、类型为t_user、id为1
http://localhost:9200/my_index/my_type/1
{
    "name" :   "zhangsan",
    "height" : "170cm",
    "age" :    25,
    "firend":  [ "lisi", "wangwu" ]
}
```

删除文档：DELETE

```
http://localhost:9200/my_index/my_type/1
```

全量更新文档：POST，注意会直接替换原文档，因此未变的字段也要加上

```
http://localhost:9200/my_index/my_type/1
{
    "name": "李四"
}
```

局部更新文档：POST，需要ES开启脚本支持(实际上还是重建)

```
http://localhost:9200/my_index/my_type/1/_update
{
    "script": {
      "lang": "painless",
      -- 脚本语句ctx._source表示当前文档，params.name指定参数
      "source":"ctx._source.name = params.name;ctx._source.sex = params.sex"
```

```
      -- 脚本参数
      "params": {
          "name" : "lisi",
          "sex" : 20
      }
    }
}
-- ctx._source.字段 = 10       // 赋值
-- ctx._source.字段 += 1       // 计算
-- ctx._source.remove('字段')  // 删除属性
-- 更多：https://blog.csdn.net/weixin_39723544/article/details/108862512
```

搜索操作

SearchRequest用于与搜索文档、聚合、定制查询有关的任何操作，还提供了在查询结果的基于上，对于匹配的关键词进行突出显示的方法。

构建请求

查询请求：非搜索，而是根据id准确查询

Java

```java
// 精准查询
GetRequest request = new GetRequest("索引","1");
// 设置包含或排除的字段信息
FetchSourceContext context = new FetchSourceContext(true, new String[]{"name", "*data"}, new String[]{});
request.fetchSourceContext(context);
try {
    GetResponse response = client.get(request, RequestOptions.DEFAULT);
    if (!response.isExists()){
        return;
    }
    // 获取结果
    String index = response.getIndex();
    String type = response.getType();
    String id = response.getId();
    long version = response.getVersion();
    DocumentField name = response.getField("name");
    Map<String, Object> sourceAsMap = response.getSourceAsMap();
} catch (IOException e) {
    e.printStackTrace();
}
```

搜索请求

Java

```java
// 创建搜索请求
SearchRequest request = new SearchRequest();
// 创建搜索请求，并指定使用的索引
SearchRequest request = new SearchRequest("索引1","索引2");

// 查询配置
request.indices("索引1","索引2", "索引3"); // 指定要搜索的索引
request.types("type1", "type2", "type3");      // 指定查询的文档类型
request.routing("routing");                    // 指定查询的路由分片
request.preference("_local");                  // 指定先查询的路由分片
request.source(searchSourceBuilder);           // 设置查询条件
request.sort(new FieldSortBuilder("name").order(SortOrder.ASC)); // 设置排序，多个字段设置多次即可

// 执行查询操作
SearchResponse response = client.search(searchRequest, RequestOptions.DEFAULT);
```

统计请求

Java

```java
CountRequest request = new CountRequest("索引1");

// 查询配置，与搜索请求一样

CountResponse response = client.count(request, RequestOptions.DEFAULT);
long total = response.getCount();
```

异步请求：

```java
ActionListener<SearchResponse> listener = new ActionListener<SearchResponse>() {

    @Override
    public void onResponse(SearchResponse indexResponse) {

    }

    @Override
    public void onFailure(Exception e) {

    }
};
client.searchAsync(searchRequest, RequestOptions.DEFAULT,listener);
```

构建查询对象

查询参数配置

```java
SearchSourceBuilder searchBuilder = new SearchSourceBuilder();
searchBuilder.query(QueryBuilders.matchAllQuery());             // 查询全部内容
searchBuilder.query(QueryBuilders.termQuery("user", "kimchy")); // 查询含关键字字段的文档
searchBuilder.query(queryBuilder);                              // 设置查询条件
sourceBuilder.from(0);                                          // 设置起始索引位置
sourceBuilder.size(5);                                          // 设置数量
sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));     // 设置超时时间
sourceBuilder.fetchSource(false);                               // 不返回文档_source的内容
sourceBuilder.fetchSource(new String[]{}, new String[]{});      // 匹配包含字段（左）、排除字段（右）
sourceBuilder.highlighter(highlightBuilder);                    // 设置高亮查询
sourceBuilder.aggregation(aggregationBuilder);                  // 设置聚合查询

```

构建查询条件 | new 或 QueryBuilders

关键字查询

```java
TermQueryBuilder queryBuilder = QueryBuilders.termQuery("name", "张三");
TermsQueryBuilder queryBuilder = QueryBuilders.termsQuery("name", "张三","李四");
```

模糊查询：

```java
MatchQueryBuilder queryBuilder = QueryBuilders.matchQuery("user", "kimchy");
queryBuilder.fuzziness(Fuzziness.AUTO); // 开启模糊查询
queryBuilder.prefixLength(3);           // 在匹配查询上设置前缀长度选项
queryBuilder.maxExpansions(10);         // 设置最大扩展选项以控制查询的模糊过程

// 左右模糊：关键词左右添加或删除一个词后进行匹配
MatchQueryBuilder queryBuilder = QueryBuilders.fuzzyQuery("name", "张三李四").fuzziness(Fuzziness.ONE);

// 关键词作为前缀
MatchQueryBuilder queryBuilder = QueryBuilders.prefixQuery("name", "张三李四");

// 通配符 * 表示多个，?表示1个字符（不建议通配符作为前缀）
MatchQueryBuilder queryBuilder = QueryBuilders.wildcardQuery("name", "张*李?");

```

分词查询："张三李四"分为"张三"、"李四"

```java
// 查询全部字段
QueryStringQueryBuilder queryBuilder = QueryBuilders.queryStringQuery("张三李四");
// 查询自定字段
QueryStringQueryBuilder queryBuilder = QueryBuilders.queryStringQuery("张三李四").defaultField("name");
// 查询自定字段：多个
QueryStringQueryBuilder queryBuilder = QueryBuilders.queryStringQuery("张三李四").field("name1").field("name2");


// 分词模糊查询： 单个字段包含分词时匹配
MatchQueryBuilder queryBuilder = QueryBuilders.matchQuery("name", "张三李四")
// 分词模糊查询： 多个字段中某一个包含分词时匹配
MatchQueryBuilder queryBuilder = QueryBuilders.multiMatchQuery("张三李四", "name1", "name2", "name3")
```

范围查询RangeQuery

```java
//闭区间查询
RangeQueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").from("fieldValue1").to("fieldValue2");
//开区间查询，默认是true，也就是包含
RangeQueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").from("fieldValue1").to("fieldValue2").includeUpper(false).includ
//大于
RangeQueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").gt("fieldValue");
//大于等于
RangeQueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").gte("fieldValue");
//小于
RangeQueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").lt("fieldValue");
//小于等于
RangeQueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").lte("fieldValue");
```

组合条件查询boolQuery

```java
QueryBuilders queryBuilder = QueryBuilders.boolQuery()
QueryBuilders queryBuilder = QueryBuilders.boolQuery().must(queryBuilder1).must(queryBuilder2);       //文档必须完全匹配条件，相当于and
QueryBuilders queryBuilder = QueryBuilders.boolQuery().mustNot(queryBuilder1).mustNot(queryBuilder2); //文档必须不匹配条件，相当于not
QueryBuilders queryBuilder = QueryBuilders.boolQuery().should(queryBuilder1).mustNot(queryBuilder2);  //至少满足一个条件，这个文档就符合shoul
```

高亮查詢：HighlightBuilder

```java
HighlightBuilder highlightBuilder = new HighlightBuilder();  // 高亮对象
highlightBuilder.preTags("<font color='red'>");              // 设置标签前缀
highlightBuilder.postTags("</font>");                        // 设置标签后缀
highlightBuilder.field("name");                              // 设置高亮字段

List<HighlightBuilder.Field> fields = highlightBuilder.fields();     // 获取高亮字段列表
HighlightBuilder.Field field = new HighlightBuilder.Field("name");   // 创建高亮字段
highlightBuilder.fields().add(field);                                // 添加高亮字段
```

聚合查询：

```java
// AggregationBuilders 聚合查询
TermsAggregationBuilder aggregationBuilder = AggregationBuilders.count("count_objid").field("objid");   // 统计
TermsAggregationBuilder aggregationBuilder = AggregationBuilders.terms("group_objid").field("objid");   // 分组


// PipelineAggregatorBuilders 计算聚合后的字段
SumBucketPipelineAggregationBuilder houseSum = PipelineAggregatorBuilders.sumBucket("houseSum", "objid");
```

查询结果处理

Java API | 文档搜索

SpringBoot 整合

https://developer.aliyun.com/article/789449#:~:text=整合 Springboot 通过虚拟机搭建ES，这里使用的版本是 6.4.3，引入相应依赖 <dependency> <groupId>org.springframework.boot<%2FgroupId>,<artifactId>spring-boot-starter-data-elasticsearch<%2FartifactId> %23 使用 2.2.2 是为了对应ES的版本 <version>2.2.2.RELEASE<%2Fversion> <%2Fdependency>

依赖

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
    # 使用2.2.2是为了对应ES的版本
    <version>2.2.2.RELEASE</version>
</dependency>
```

配置

```yaml
spring:
  data:
    elasticsearch:
      # 在es中配置的名称
      cluster-name: es
      # 如果是集群，用,分隔
      cluster-nodes: 192.168.1.7:9300
```

客户端创建 PreBuiltTransportClient

- 客户端文档：https://www.elastic.co/guide/en/elasticsearch/client/index.html

javaAPI本质上是java代码作为客户端向ES服务发起请求，客户端pom依赖如下：

```xml
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
    <version>6.2.4</version>
</dependency>
```

创建客户端

```java
private TransportClient client;

@BeforeEach
public void test1() throws UnknownHostException {
    // 指定节点
    Settings settings = Settings.builder().put("cluster.name", "myes").build();

    // 指定节点信息
    client = new PreBuiltTransportClient(settings)
                .addTransportAddress(new TransportAddress(InetAddress.getByName("node01"),9300))
                .addTransportAddress(new TransportAddress(InetAddress.getByName("node02"),9300));
}
```

创建索引

```java
public void createIndex(){



    String json = "...";
    // 使用json创建索引
    IndexResponse index = client.prepareIndex("db", "user", "1").setSource(json, XContentType.JSON).get();
    // 使用map创建索引
    IndexResponse index = client.prepareIndex("db", "user", "2").setSource(jsonMap).get();
    // 使用XcontentBuilder创建索引
    IndexResponse index = client.prepareIndex("db", "user", "3")
            .setSource(new XContentFactory().jsonBuilder()
                    .startObject()
                    .field("name", "lisi")
                    .field("age", "18")
                    .field("sex", "0")
                    .field("address", "bj")
                    .endObject())
            .get();

     // 批量创建索引
     BulkRequestBuilder bulk = client.prepareBulk();
      bulk.add(client.prepareIndex("db1", "user", "4")
              .setSource(new XContentFactory().jsonBuilder()
                      .startObject()
                      .field("name", "wangwu")
                      .field("age", "18")
                      .field("sex", "0")
                      .field("address", "bj")
```

```
                    .endObject()));
        bulk.add(client.prepareIndex("db2", "user", "5")
                .setSource(new XContentFactory().jsonBuilder()
                        .startObject()
                        .field("name", "zhaoliu")
                        .field("age", "18")
                        .field("sex", "0")
                        .field("address", "bj")
                        .endObject()));
        BulkResponse bulkResponse = bulk.get();

        // 更新索引
        client.prepareUpdate().setIndex("db").setType("user").setId("8").setDoc(json,XContentType.JSON).get();

        // 删除索引
        DeleteResponse delete = client.prepareDelete("db", "user", "8").get();
        // 删除整个索引库
        DeleteIndexResponse delete = client.admin().indices().prepareDelete("db").execute().actionGet();


        // 创建映射
        XContentBuilder mapping = XContentFactory.jsonBuilder()
                .startObject()
                .startObject("properties")
                .startObject("id").field("type", "integer").endObject()
                .startObject("name").field("type", "text").field("analyzer", "ik_max_word").endObject();
        //pois：索引名   cxyword：类型名（可以自己定义）
        PutMappingRequest putmap = Requests.putMappingRequest("db").type("user").source(mapping);
        //为索引添加映射
        client.admin().indices().putMapping(putmap).actionGet();


        // 通过id11精确查询
        GetResponse get = client.prepareGet("db", "user", "11").get();
        String index = get.getIndex();
        String type = get.getType();
        String id = get.getId();
        Map<String, Object> source = get.getSource();

        // 查询所有数据（详细查看文档搜索）
        SearchResponse response = client.prepareSearch("db").setTypes("user").setQuery(new s()).get();
        SearchHits searchHits = response.getHits();
        SearchHit[] hits = searchHits.getHits();
        for (SearchHit hit : hits) {
            String sourceAsString = hit.getSourceAsString();
            System.out.println(sourceAsString);
        }


        client.close();
    }
```

客户端创建 RestHighLevelClient

操作：

https://www.cnblogs.com/leeSmall/p/9218779.html

创建索引

Java

```
public void createIndex(RestHighLevelClient client) {

        // 1、创建 创建索引request 参数：索引名mess
        CreateIndexRequest request = new CreateIndexRequest("mess");

        // 2、设置索引的settings
        request.settings(Settings.builder().put("index.number_of_shards", 3) // 分片数
                .put("index.number_of_replicas", 2) // 副本数
                .put("analysis.analyzer.default.tokenizer", "ik_smart") // 默认分词器
        );

        // 3、设置索引的mappings
        request.mapping("_doc",
                "  {\n" +
                "    \"_doc\": {\n" +
                "      \"properties\": {\n" +
```

```
                "        \"message\": {\n" +
                "          \"type\": \"text\"\n" +
                "        }\n" +
                "      }\n" +
                "    }\n" +
                "  }",
                XContentType.JSON);

        // 4、 设置索引的别名
        request.alias(new Alias("mmm"));

        // 5、 发送请求
        // 5.1 同步方式发送请求
        CreateIndexResponse createIndexResponse = client.indices()
                .create(request);

        // 6、处理响应
        boolean acknowledged = createIndexResponse.isAcknowledged();
        boolean shardsAcknowledged = createIndexResponse
                .isShardsAcknowledged();
        System.out.println("acknowledged = " + acknowledged);
        System.out.println("shardsAcknowledged = " + shardsAcknowledged);

        // 5.1 异步方式发送请求
        /*ActionListener<CreateIndexResponse> listener = new ActionListener<CreateIndexResponse>() {
            @Override
            public void onResponse(
                    CreateIndexResponse createIndexResponse) {
                // 6、处理响应
                boolean acknowledged = createIndexResponse.isAcknowledged();
                boolean shardsAcknowledged = createIndexResponse
                        .isShardsAcknowledged();
                System.out.println("acknowledged = " + acknowledged);
                System.out.println(
                        "shardsAcknowledged = " + shardsAcknowledged);
            }

            @Override
            public void onFailure(Exception e) {
                System.out.println("创建索引异常：" + e.getMessage());
            }
        };

        client.indices().createAsync(request, listener);
        */
}
```

添加文档

Java

```
public void addData(RestHighLevelClient client) {
        // 1、创建索引请求
        IndexRequest request = new IndexRequest(
                "mess",   //索引
                "_doc",     // mapping type
                "1");     //文档id

        // 2、准备文档数据
        // 方式一：直接给JSON串
        String jsonString = "{" +
                "\"user\":\"kimchy\"," +
                "\"postDate\":\"2013-01-30\"," +
                "\"message\":\"trying out Elasticsearch\"" +
                "}";
        request.source(jsonString, XContentType.JSON);

        // 方式二：以map对象来表示文档
        /*
        Map<String, Object> jsonMap = new HashMap<>();
        jsonMap.put("user", "kimchy");
        jsonMap.put("postDate", new Date());
        jsonMap.put("message", "trying out Elasticsearch");
        request.source(jsonMap);
        */

        // 方式三：用XContentBuilder来构建文档
        /*
        XContentBuilder builder = XContentFactory.jsonBuilder();
        builder.startObject();
```

```
{
    builder.field("user", "kimchy");
    builder.field("postDate", new Date());
    builder.field("message", "trying out Elasticsearch");
}
builder.endObject();
request.source(builder);
*/

// 方式四：直接用key-value对给出
/*
request.source("user", "kimchy",
                "postDate", new Date(),
                "message", "trying out Elasticsearch");
*/

//3、其他的一些可选设置
/*
request.routing("routing");  //设置routing值
request.timeout(TimeValue.timeValueSeconds(1));  //设置主分片等待时长
request.setRefreshPolicy("wait_for");  //设置重刷新策略
request.version(2);  //设置版本号
request.opType(DocWriteRequest.OpType.CREATE);  //操作类别
*/

//4、发送请求
IndexResponse indexResponse = null;
try {
    // 同步方式
    indexResponse = client.index(request);
} catch(ElasticsearchException e) {
    // 捕获，并处理异常
    //判断是否版本冲突、create但文档已存在冲突
    if (e.status() == RestStatus.CONFLICT) {
        logger.error("冲突了，请在此写冲突处理逻辑！\n" + e.getDetailedMessage());
    }

    logger.error("索引异常", e);
}

//5、处理响应
if(indexResponse != null) {
    String index = indexResponse.getIndex();
    String type = indexResponse.getType();
    String id = indexResponse.getId();
    long version = indexResponse.getVersion();
    if (indexResponse.getResult() == DocWriteResponse.Result.CREATED) {
        System.out.println("新增文档成功，处理逻辑代码写到这里。");
    } else if (indexResponse.getResult() == DocWriteResponse.Result.UPDATED) {
        System.out.println("修改文档成功，处理逻辑代码写到这里。");
    }
    // 分片处理信息
    ReplicationResponse.ShardInfo shardInfo = indexResponse.getShardInfo();
    if (shardInfo.getTotal() != shardInfo.getSuccessful()) {

    }
    // 如果有分片副本失败，可以获得失败原因信息
    if (shardInfo.getFailed() > 0) {
        for (ReplicationResponse.ShardInfo.Failure failure : shardInfo.getFailures()) {
            String reason = failure.reason();
            System.out.println("副本失败原因：" + reason);
        }
    }
}


//异步方式发送索引请求
/*ActionListener<IndexResponse> listener = new ActionListener<IndexResponse>() {
    @Override
    public void onResponse(IndexResponse indexResponse) {

    }

    @Override
    public void onFailure(Exception e) {

    }
};
client.indexAsync(request, listener);
```

```
     */
}
```

查询文档

Java

```
public void searchData(RestHighLevelClient client) {
      // 1、创建获取文档请求
      GetRequest request = new GetRequest(
              "mess",   //索引
              "_doc",     // mapping type
              "1");     //文档id

      // 2、可选的设置
      //request.routing("routing");
      //request.version(2);

      //request.fetchSourceContext(new FetchSourceContext(false)); //是否获取_source字段
      //选择返回的字段
      String[] includes = new String[]{"message", "*Date"};
      String[] excludes = Strings.EMPTY_ARRAY;
      FetchSourceContext fetchSourceContext = new FetchSourceContext(true, includes, excludes);
      request.fetchSourceContext(fetchSourceContext);

      //也可写成这样
      /*String[] includes = Strings.EMPTY_ARRAY;
      String[] excludes = new String[]{"message"};
      FetchSourceContext fetchSourceContext = new FetchSourceContext(true, includes, excludes);
      request.fetchSourceContext(fetchSourceContext);*/


      // 取stored字段
      /*request.storedFields("message");
      GetResponse getResponse = client.get(request);
      String message = getResponse.getField("message").getValue();*/


      //3、发送请求
      GetResponse getResponse = null;
      try {
          // 同步请求
          getResponse = client.get(request);
      } catch (ElasticsearchException e) {
          if (e.status() == RestStatus.NOT_FOUND) {
              logger.error("没有找到该id的文档" );
          }
          if (e.status() == RestStatus.CONFLICT) {
              logger.error("获取时版本冲突了，请在此写冲突处理逻辑！" );
          }
          logger.error("获取文档异常", e);
      }

      //4、处理响应
      if(getResponse != null) {
          String index = getResponse.getIndex();
          String type = getResponse.getType();
          String id = getResponse.getId();
          if (getResponse.isExists()) { // 文档存在
              long version = getResponse.getVersion();
              String sourceAsString = getResponse.getSourceAsString(); //结果取成 String
              Map<String, Object> sourceAsMap = getResponse.getSourceAsMap();  // 结果取成Map
              byte[] sourceAsBytes = getResponse.getSourceAsBytes();    //结果取成字节数组

              logger.info("index:" + index + "  type:" + type + "  id:" + id);
              logger.info(sourceAsString);

          } else {
              logger.error("没有找到该id的文档" );
          }
      }


      //异步方式发送获取文档请求
      /*
      ActionListener<GetResponse> listener = new ActionListener<GetResponse>() {
          @Override
          public void onResponse(GetResponse getResponse) {
```

```
        }

        @Override
        public void onFailure(Exception e) {

        }
    };
    client.getAsync(request, listener);
    */

  }
```

文档搜索

更多：https://www.cnblogs.com/reycg-blog/p/9946821.html

RestFull API | 接口操作

https://blog.csdn.net/ctwy291314/article/details/81202386

文档操作

文档操作 | 搜索

根据文档范围查询文档：

```ada
-- 查询所有索引
/_search

-- 查询索引db1，db2所有类型
/db1,db2/_search

-- 查询db开头的索引
/db*/_search

-- 查询指定文档指定类型
/db/user,tweet/_search

-- 查询所有文档中指定类型
/_all/user,tweet/_search
```

根据文档内容查询文档：head、get

```ada
-- 检查文档是否存在
HEAD /testdb/t_user/1

-- 获取一条文档内容
GET /testdb/t_user/1

-- _search搜索一次文档内容：默认返回10条
GET /testdb/t_user/_search

-- 高亮搜索（条件查询）：作为条件的字段即为高亮
GET /testdb/t_user/_search?q=name:zhangsan
GET /testdb/t_user/_search?q=name:zhangsan+age=20  -- 多条件

-- 分页搜索：from为开始条数， 默认为0
GET /_search?size=5&from=10
```

请求体查询：使用DSL语言查询：JSON请求体

```ada
```

```
GET /testdb/t_user/_search

-- 空查询
{}

-- 分页查询
{
  "from": 30,  -- 起始文档
  "size": 10
}

-- 条件查询
{
    -- 搜索条件
    "query" : {
        -- match： 模糊查询，相关式（与mysql查询不同，不是包含关系，而是根据关联程度搜索并排序，比如可搜索出“zhang”）
        "match" : {
            "name" : "zhangsan"    -- 查询条件
        },
        -- match_phrase: 模糊查询，包含式
        "match_phrase" : {
            "name" : "zhangsan"
        }
    },

    -- 分组聚合查询
    "aggs": {
        -- 结果名称
        "names": {
            -- terms： 统计数量
            "terms": { "field": "name" }, -- 聚合的字段: 按照name分组
             -- 嵌套一层分组聚合
            "aggs" : {
                "avgs": {
                    -- avg：求平均值
                    "avg" : { "field" : "age" }
                }
             }

         },

    }

    -- 高亮结果：将搜索结果中匹配的内容加亮，实际上以 HTML 标签 <em></em> 封装匹配内容
    "highlight": {
        -- fields指定需要高亮的筛选字段
        "fields" : {
            "name" : {}
        }
    }
}

-- 条件查询2
{
    "query" : {
        "bool": {
            "must": {
                "match" : {
                    "name" : "zhangsan"
                }
            },
            "filter": {
                -- range： 范围查询
                "range" : {
                    "age" : { "gt" : 30 }  -- 查询条件，gt代表大于
                }
            }
        }
    }
}
```

- 深入搜索：https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-in-depth.html

分析

倒排索引：https://www.elastic.co/guide/cn/elasticsearch/guide/current/inverted-index.html

将文本文档进行分词，并标记每个词分别在哪些文档出现，不过需要研究同义词的问题。

分布式集群

查看Es版本

```Ada
http://localhost:9200
```

集群健康检查

```Ada
http://localhost:9200/_cat/health?v
```

分布式存储 | 分片

分片属于索引，它可以存放在不同节点主机中，并且均可设置副本分片，文档路由到分片的规则：

shard = hash(routing) % number_of_primary_shards

- routing是一个可变值，默认是文档的 `_id`
- number_of_primary_shards 是主分片数量
- shard是计算出文档的位置信息

添加文档时，根据文档id来计算出文档存储在拿个分片；操作文档时根据文档id计算文档被存在哪个分片。

设置分片

```Ada
http://localhost:9200/索引名称（小写）

-- put参数

{
    "settings": {
    "number_of_shards" :   1,     -- 主分片数量
    "number_of_replicas" : 0      -- 副本数量
    }
}


-- 修改分片副本数量
http://localhost:9200/索引名称/_settings

--put 参数
{
    "number_of_replicas": 2
}

-- 查询索引信息
http://localhost:9200/索引名称/_search_shards
```

ES工具面板

https://www.jianshu.com/p/54e04b5b5ce2

#

- 字符串: `string`
- 整数： `byte`, `short`, `integer`, `long`
- 浮点数: `float`, `double`

- 布尔型: `boolean`
- 日期: `date`

pickle 模块

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过pickle模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

注解： 从 file 中读取一个字符串，并将它重构为原来的python对象。

file: 类文件对象，有read()和readline()接口。

完整的语法格式为：

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

参数说明:

- file: 必需，文件路径（相对或者绝对路径）。
- mode: 可选，文件打开模式
- buffering: 设置缓冲
- encoding: 一般使用utf8
- errors: 报错级别
- newline: 区分换行符
- closefd: 传入的file参数类型
- opener: