ElasticSearch

1、介绍

> 全文检索（MyISAM支持， Innodb）

1.1 海量数据检索

> 对于海量数据进行检索如果采用MySQL，效率太低

1.2 全文检索

> 在海量数据中执行搜索功能时，如果使用MySQL，无法实现

1.3 实时性

> 在实现生成的数据中，需要立马检索出来

1.4 高亮显示

> 将检索出的结果需要进行突出

2、ES概述

> - 用java写的基于lucene的一款全文检索框架
> - 源码开放，搜索实时，分布式
> - 对外提供的接口符合RESTFull风格

> ES和Solr
>
> - 都是基于Lucene
> - Solr查询离线数据速度会比较快，如果查询实时数据ES比较快
> - Solr的集群需要Zookeeper进行管理，ES自带有管理组件
> - 在大数据中ES比Solr更有市场

3、ES的安装

3.1 安装ElasticSearch和Kibana

> 创建Docker-compose.yml

```
version: "3.1"
services:
  elasticsearch:
    image: daocloud.io/library/elasticsearch:6.5.4
    restart: always
    container_name: elasticsearch
    ports:
      - 9200:9200
  kibana:
    image: daocloud.io/library/kibana:6.5.4
    restart: always
    container_name: kibana
    ports:
      - 5601:5601
    environment:
      - elasticsearch_url=http://192.168.136.129:9200
    depends_on:
      - elasticsearch
```

3.2 安装可能出现的错误

> 注意：
>
> - 默认在虚拟机中启动es会出现虚拟内存不足错误！

```
elasticsearch    | [2020-08-20T02:18:50,400][INFO ][o.e.b.BootstrapChecks    ]
[toN2lHJ] bound or publishing to a non-loopback address, enforcing bootstrap
checks
elasticsearch    | ERROR: [1] bootstrap checks failed
elasticsearch    | [1]: max virtual memory areas vm.max_map_count [65530] is too
low, increase to at least [262144]
```

```
临时解决办法（重启虚拟机会失效）：
1.切换到root用户，执行命令：
    sysctl -w vm.max_map_count=262144
2.查看结果：
    sysctl -a|grep vm.max_map_count
3.显示：
    vm.max_map_count = 262144
永久解决办法
    在/etc/sysctl.conf文件最后添加一行：vm.max_map_count=262144
    重启虚拟机
```

> - jvm的内存不足

```
[root@localhost docker_elasticsearch]# find /var/lib/docker/ -name jvm.options
/var/lib/docker/overlay2/e8529a58709388e69d66a4ed3352108afa7dc85d87821e95c938eeb
eea83bddf/merged/usr/share/elasticsearch/config/jvm.options
/var/lib/docker/overlay2/2aca90eba05bc46365e432797088581ac20f46535b38d8a36c3fbda
57e6b5923/diff/usr/share/elasticsearch/config/jvm.options
```

> 验证es是否安装成功
>
> 在浏览器中输入地址：http://192.168.136.129:9200/
>
> 
>

> 访问kibana:http://192.168.136.129:5601/

3.3 安装IK分词器

> 关键词：我是中国人-> 我　中国 中国人 国人
>
> github上的下载地址：https://github.com/medcl/elasticsearch-analysis-ik/archive/v6.5.4.zip
>
> 采用国内服务：http://bishe.itluma.cn:8080/files/elasticsearch-analysis-ik-6.5.4.zip
>
> - 进入es内部安装该插件

```
[root@localhost docker_elasticsearch]# docker exec -it elasticsearch bash
[root@85d858270e8f elasticsearch]# ls
LICENSE.txt  NOTICE.txt  README.textile  bin  config  data  lib  logs  modules
plugins
[root@85d858270e8f elasticsearch]# cd bin/
[root@85d858270e8f bin]# ./elasticsearch-plugin  install
http://bishe.itluma.cn:8080/files/elasticsearch-analysis-ik-6.5.4.zip
-> Downloading http://bishe.itluma.cn:8080/files/elasticsearch-analysis-ik-
6.5.4.zip
[=================================================] 100%??
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

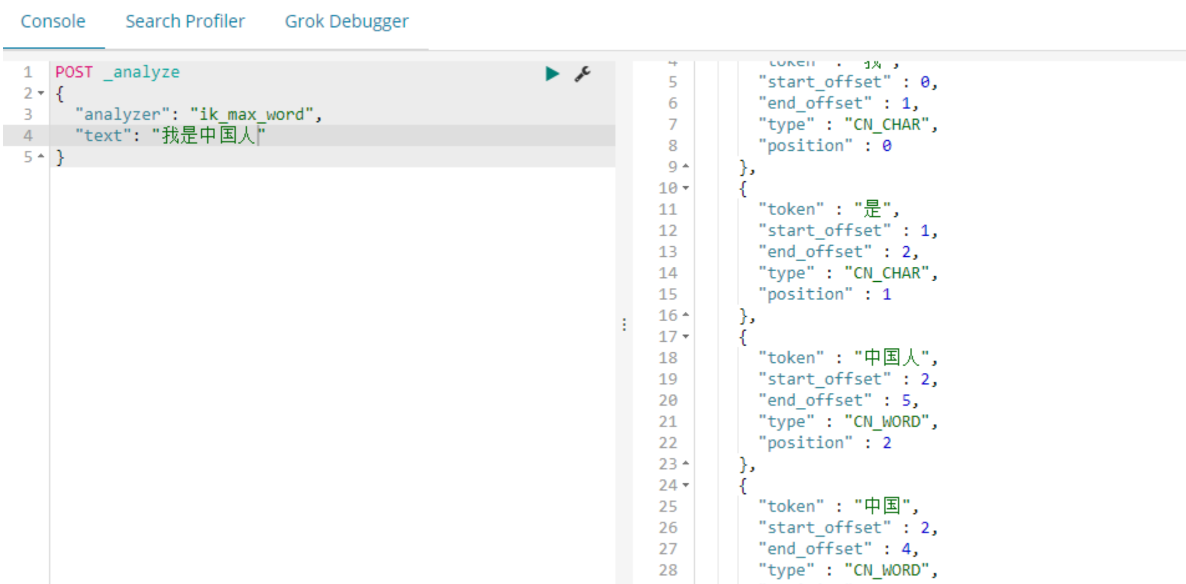
```
@     WARNING: plugin requires additional permissions     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
* java.net.SocketPermission * connect,resolve
See
http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html
for descriptions of what these permissions allow and the associated risks.

Continue with installation? [y/N]y
-> Installed analysis-ik
```

> 安装好之后重启es

```
[root@localhost docker_elasticsearch]# docker restart elasticsearch
elasticsearch
```

> 校验分词器



4、ES的基本操作(重点)

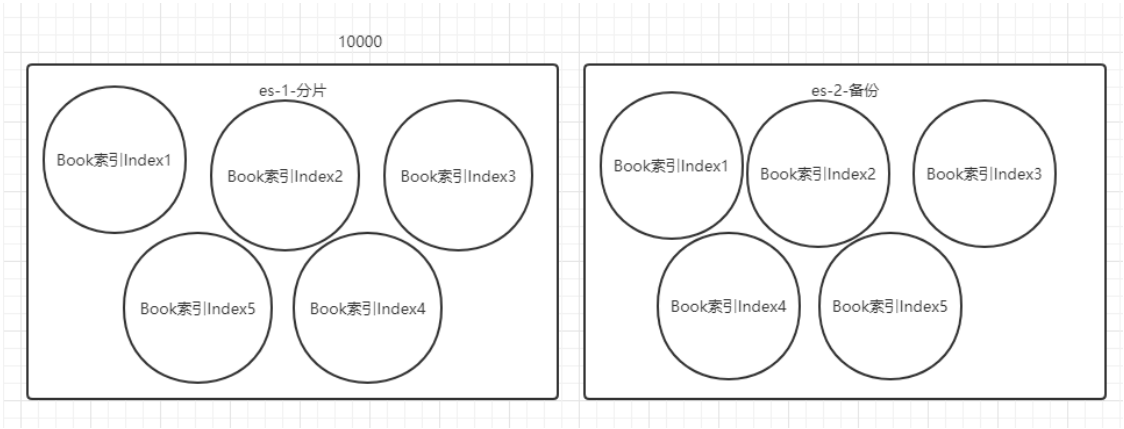
4.1 ES的结构

4.1.1 概念

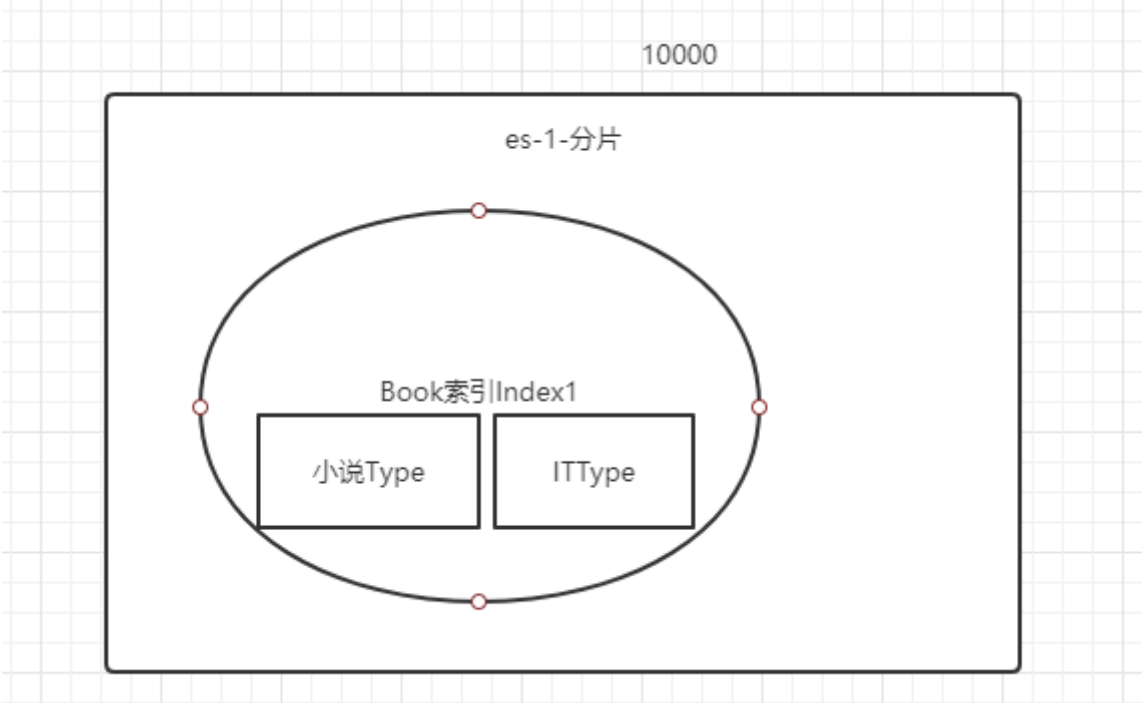
> - 分片（shards）:将索引进行拆分，提高扩容能力
> - 副本(replicas):对分片进行备份，提高可靠性
> - 索引(index):类似于数据库中的库的概念，存放相似特征的文档集合
> - 类型(type):类似于数据库中的表的概念，在一个索引中可以定义多个类型
> - 字段(field):类似于数据库中表的字段概念
> - 文档(document):类似于数据库中的行的概念，是索引的基础信息单元

4.1.2 索引：index，主要是分成分片和副本

> - 默认每个索引被分成5个分片
> - 每一个分片至少存储一个副本，也就是至少需要两台服务器存储
> - 默认副本是不会进行数据检索的，当分片的检索压力过大时才会进行数据检索

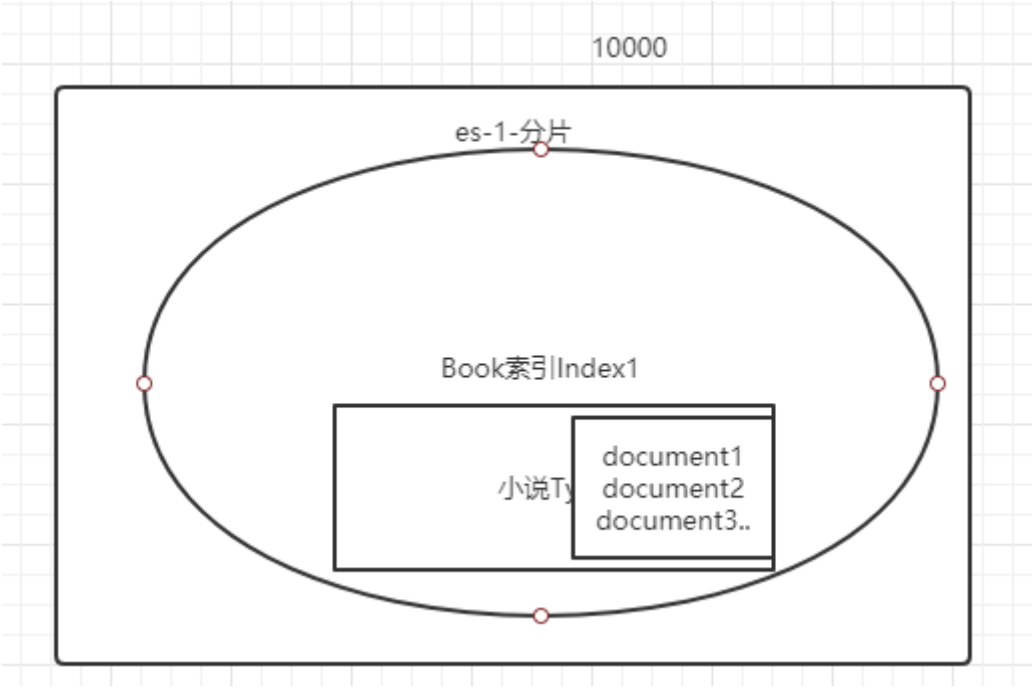


4.1.3 类型：一个索引下，可以创建多个类型（根据不同的版本，类型的创建也有变化）

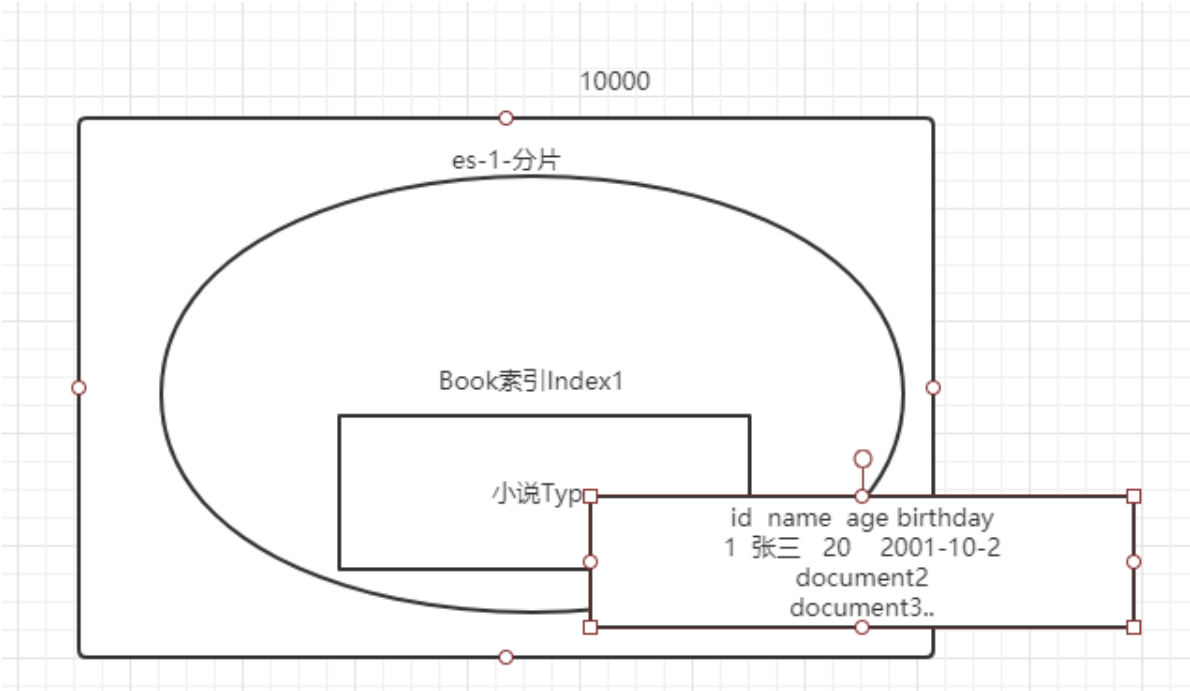


> 注意：因为ElasticSearch版本的迭代将Type类型进行了替换
>
> - 5.x的版本的，一个index下可以创建多个Type
> - 6.x的版本中，一个index下只能创建一个Type
> - 7.x的版本中，一个index下没有了Type

4.1.4 文档(document)：一个类型下，可以有多个文档，类似于数据库中的行的概念



4.4.5 字段(field)：在一个文档中，可以包含多个属性，类似于数据库中的列的概念



4.2 操作ES的RESTFul接口

> - GET请求
>   - http://ip:port/index 查询索引
>   - http://ip:port/index/type/doc_id 查询具体的文档索引
>   - http://ip:port/index/type/_search：查询文档，可以在请求体添加json添加查询
> - POST请求
>   - http://ip:port/index/type/_update：修改文档，在请求体中添加json格式的修改条件
> - PUT请求
>   - http://ip:port/index:创建一个索引，需要在请求体中指定索引信息
>   - http://ip:port/index/type/_mappings :创建索引，指定索引的文档中存储属性信息
> - DELETE请求

- http://ip:port/index:删除所有
- http://ip:port/index/type/doc_id:删除指定文档

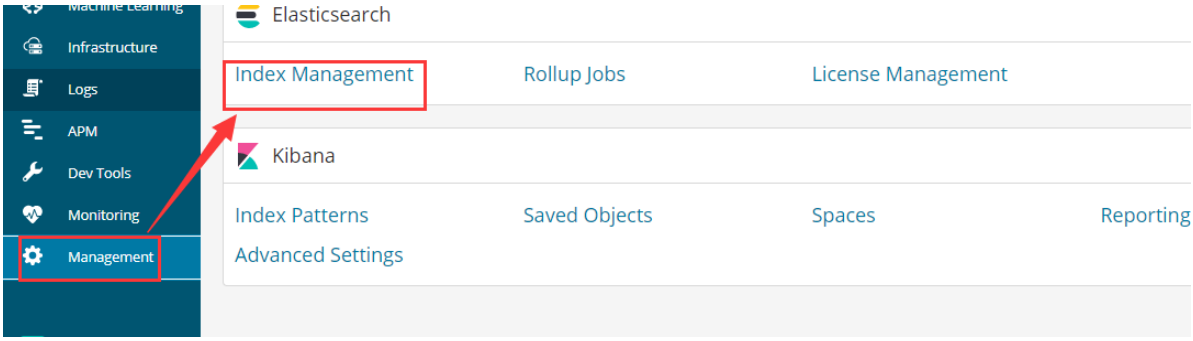
4.3 索引操作

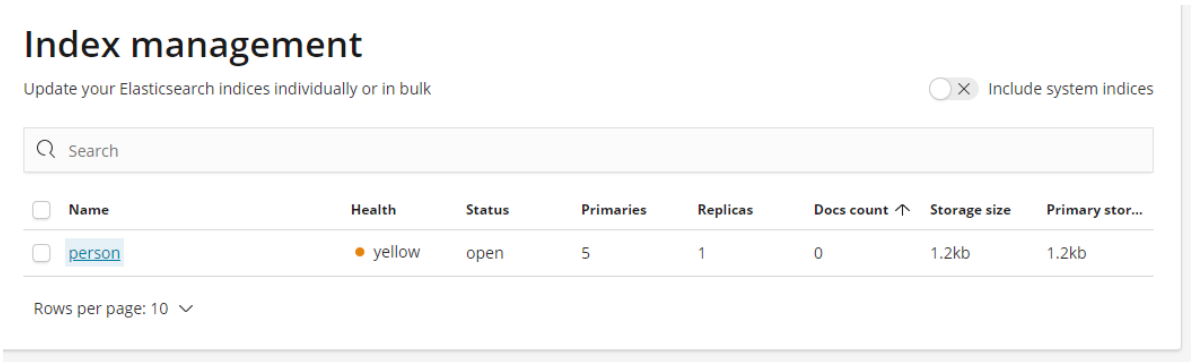
4.3.1 创建索引

```
PUT /person
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1
  }
}
```

4.3.2 查询索引

```
GET /person
```





4.3.3 删除索引

```
DELETE /person
```

4.4 ES中字段的类型

官方文档：https://www.elastic.co/guide/en/elasticsearch/reference/6.5/mapping-types.html

- 字符串类型：
  - text:可以用于全文检索，进行分词
  - keyword:不被进行分词
- 数值类型
  - long:占8个字节
  - integer：占4个字节
  - short：占2个字节

    - byte：占1个字节
    - double：占8个字节
    - float：4个字节
    - half_float ： 2个字节
    - scaled_float :根据一个long和scaled来表达一个浮点类型 long 123 scaled:100->1.23
- 时间类型
    - date
- 布尔类型
    - booleanl类型，true/false
- 二进制类型
    - binary:但是存储是需要将二进制转换成Base64编码格式的字符串
- 范围类型greater than/less than equals
    - integer_range:赋值的时候,,不需要给定具体的值，给一个integer范围就可以了：gte/lte/gt/lt
    - float_range:
    - long_range：
    - double_range:
    - date_range:
    - ip_range:
- 经纬度类型：
    - geo-point:存储经纬度
- ip类型：
    - ip:可以存Ipv4或者Ipv6

4.5 创建索引并指定字段类型

```
PUT /book
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1
  },
  "mappings": {
    "novel":{
      "properties":{
        "name":{
          "type":"text",
           "analyzer":"ik_max_word",
           "index":true,
           "store":false
        },
        "author":{
          "type":"keyword"
        },
        "price":{
          "type":"long"
        },
        "count":{
          "type":"long",
          "index":false
        },
        "pubdate":{
          "type":"date",
```

```
            "format":"yyyy-MM-dd HH:mm:ss"
          },
          "decr":{
            "type":"text",
            "analyzer":"ik_max_word"
          }
        }
      }
    }
}
```

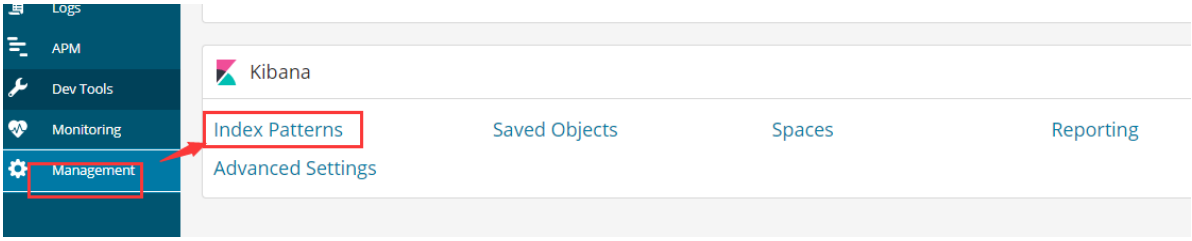
4.6 文档操作

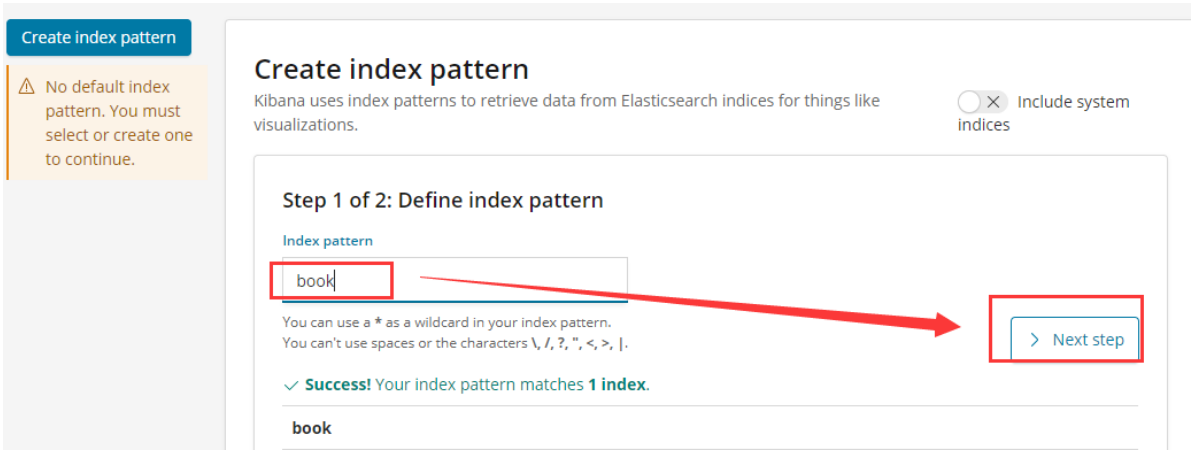
> 在es服务中唯一标识符，`_index`， `_type`， `_id`三个内容的组合，来确定一个具体的文档

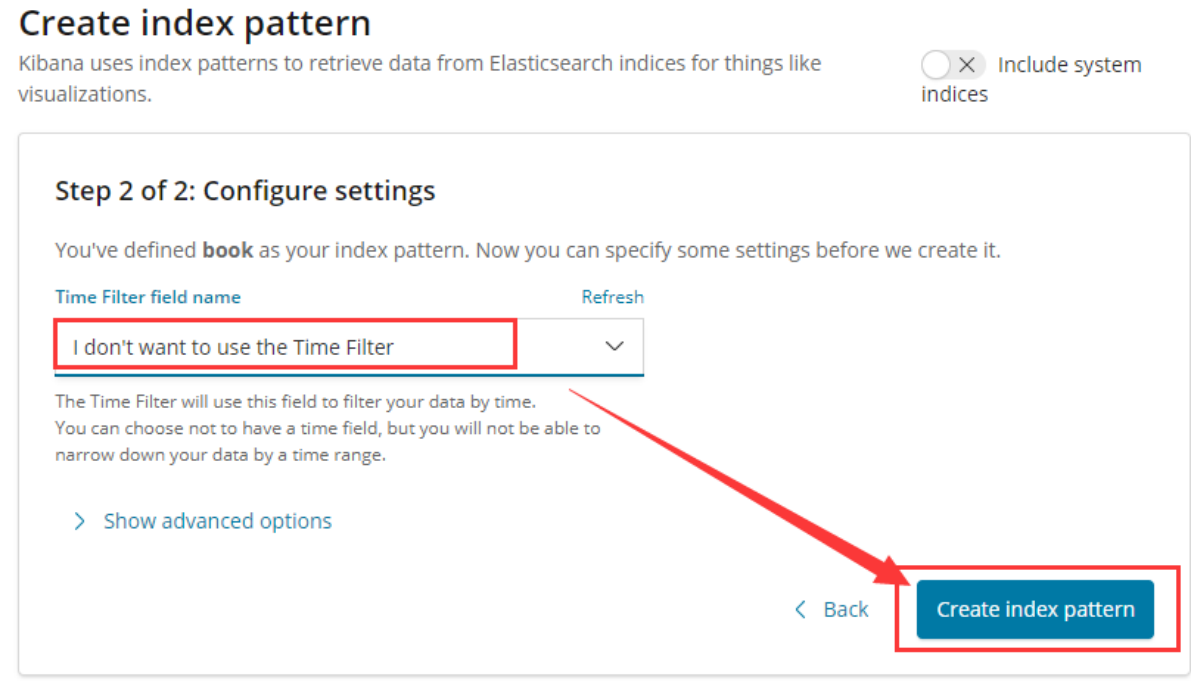
4.6.1 添加文档

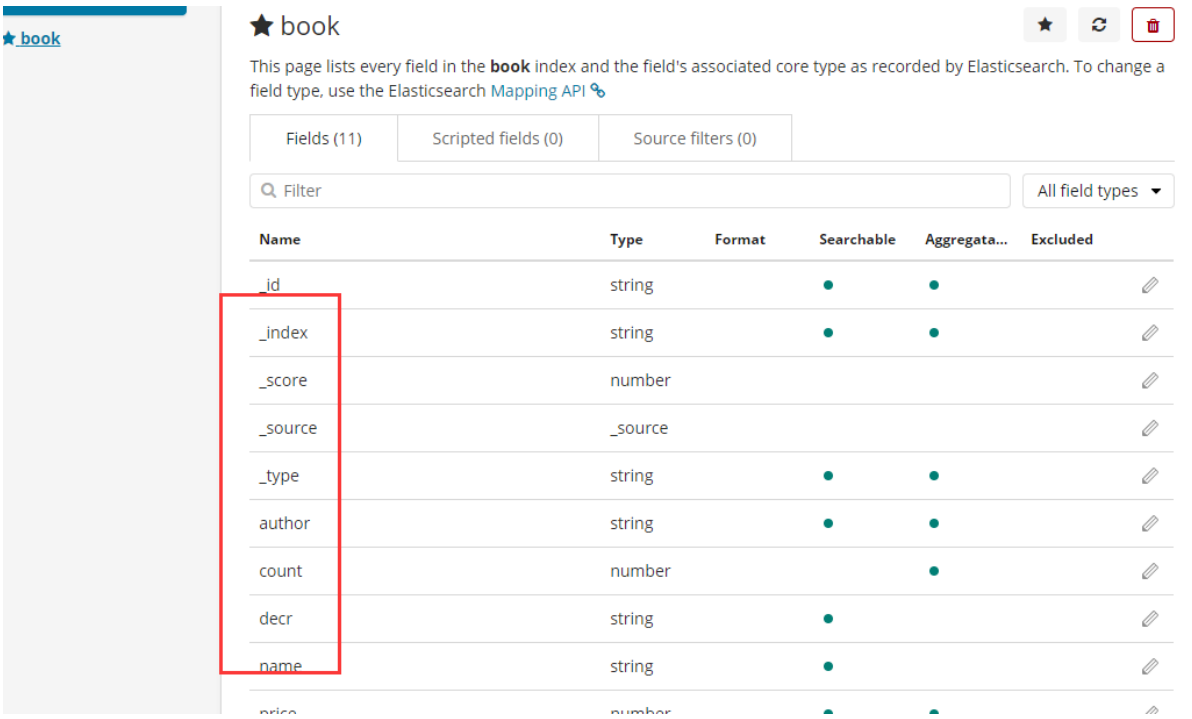
> id自动生成

```
#添加文档，自动生成文档id
POST /book/novel
{
    "name":"西游记",
    "author":"吴承恩",
    "price":8888,
    "count":100,
    "pubdate":"2020-8-19 12:12:12",
    "decr":"你这泼猴，我要再压你五百年！"
}
```






Create index pattern
Kibana uses index patterns to retrieve data from Elasticsearch indices for things like visualizations.
Include system indices
Step 2 of 2: Configure settings
You've defined book as your index pattern. Now you can specify some settings before we create it.
Time Filter field name
Refresh
I don't want to use the Time Filter
The Time Filter will use this field to filter your data by time.
You can choose not to have a time field, but you will not be able to narrow down your data by a time range.
Show advanced options
Back
Create index pattern


book
book
This page lists every field in the book index and the field's associated core type as recorded by Elasticsearch. To change a field type, use the Elasticsearch Mapping API
Fields (11)
Scripted fields (0)
Source filters (0)
Filter
All field types
Name
Type
Format
Searchable
Aggregata...
Excluded
_id string
_index string
_score number
_source _source
_type string
author string
count number
decr string
name string


Discover
Visualize
Dashboard
Timelion
Canvas
Machine Learning
Infrastructure
Logs
APM
Yes
No
New
Save
Open
Share
Inspect
Auto-refresh
Options
Refresh
Add a filter
book
Selected fields
_source
Available fields
_id
_source

> 手动添加id

```
#添加文档，手动指定id
POST /book/novel/2
{
    "name":"三国演义",
    "author":"罗贯中",
    "price":2789,
    "count":100,
    "pubdate":"2020-8-19 12:12:12",
    "decr":"再活五百年，我就成仙了"
}
```

4.6.2 修改文档

> 指定文档id

```
PUT /book/novel/3
{
   "name":"红楼梦",
    "author":"曹雪芹",
    "price":1000,
    "count":100,
    "pubdate":"2020-8-19 12:12:12",
  "decr":"美女和野兽"
}
```

> 基于doc方式进行修改

```
#_update表示修改
POST /book/novel/3/_update
{
   "doc":{
      #指定需要修改的字段 字段名：对应的值
     "decr":"假作真时真亦假"
   }
}
```

4.6.3 删除文档

> 根据id删除

```
DELETE /book/novel/_id
```

5、Java操作ES

5.1 Java操作索引

5.1.1 Java连接ES

> 创建maven工程

```
<dependencies>
    <!--1、elasticsearch-->
    <dependency>
        <groupId>org.elasticsearch</groupId>
        <artifactId>elasticsearch</artifactId>
```

```
        <version>6.5.4</version>
    </dependency>

    <!--2、操作elasticsearch高级API-->
    <dependency>
        <groupId>org.elasticsearch.client</groupId>
        <artifactId>elasticsearch-rest-high-level-client</artifactId>
        <version>6.5.4</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12-beta-3</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>
</dependencies>
```

> 连接与es的连接

```
//创建es的服务器对象
HttpHost host = new HttpHost("192.168.136.129", 9200);

//获取builder对象
RestClientBuilder clientBuilder = RestClient.builder(host);

//构造客户端对象
RestHighLevelClient client = new RestHighLevelClient(clientBuilder);
```

5.1.2 生成索引

> 通过RestClient在es中生成索引

```
//操作es
//构建索引
//1.准备索引的settings参数
Settings.Builder settings =
Settings.builder().put("number_of_shards",5).put("number_of_replicas",1);

//2.准备关于索引的mapping参数
XContentBuilder mappings = JsonXContent.contentBuilder().
    startObject().
        startObject("properties").
            startObject("name").
                field("type", "text").field("analyzer", "ik_max_word").
            endObject().
            startObject("author").
                field("type", "keyword").
            endObject().
            startObject("price").
                field("type", "long").
            endObject().
            startObject("pubdate").
```

```
                field("type", "date").field("format", "yyyy-MM-dd HH:mm:ss").
            endObject().
        endObject().
    endObject();

//3.将settings和mapping装配到一个request对象中
CreateIndexRequest request = new
CreateIndexRequest("book2").settings(settings).mapping("novel", mappings);

//4.通过client对象执行请求
CreateIndexResponse response = client.indices().create(request,
RequestOptions.DEFAULT);

System.out.println(response);//DOM4j POI
```

5.1.3 判断是否存在索引

```
//创建查询所有对象
GetIndexRequest request2 = new GetIndexRequest();
//设置索引名
request2.indices(indexName);
//检测索引是否存在
boolean result = client.indices().exists(request2, RequestOptions.DEFAULT);
```

5.1.4 删除索引

```
//创建删除对象
DeleteIndexRequest deleteIndexRequest = new DeleteIndexRequest();
//指定索引名称
deleteIndexRequest.indices(indexName);

//执行删除过程
AcknowledgedResponse response = client.indices().delete(deleteIndexRequest,
RequestOptions.DEFAULT);
System.out.println(response.isAcknowledged()?indexName+"删除成功！":indexName+"删除
失败！");
```

5.2 文档操作

> 定义实体类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Novel {
    @JsonInclude
    private Integer id;
    private String name;
    private String author;
    private Long price;

    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date pubDate;
}
```

5.2.1 添加文档

```
@Test
public void testAdd() throws IOException {
    //添加文档数据
    RestHighLevelClient client = ESClient.getClient();
    String indexName = "book2";
    String typeName = "novel";

    //1.准备json数据
    Novel novel = new Novel(1001, "金瓶梅","杨光", 1000L, new Date());
    ObjectMapper objectMapper = new ObjectMapper();
    String json = objectMapper.writeValueAsString(novel);
    System.out.println(json);

    //2.准备一个request对象(手动方式指定Id)
    IndexRequest request = new IndexRequest(indexName, typeName,
novel.getId().toString());
    request.source(json, XContentType.JSON);//设置文档


    //3.通过client生成文档
    IndexResponse response = client.index(request, RequestOptions.DEFAULT);


    //4.返回结果
    System.out.println(response.getResult());//CREATED
}
```

5.2.2 修改文档

```
@Test
public void testUpdate() throws IOException {

    //1.准备需要修改的集合
    Map<String,Object> doc = new HashMap<>();
    doc.put("author", "杨光光");
    doc.put("price", 1888);

    String docId = "1001";
    //2.准备一个request对象(手动方式指定Id)
    UpdateRequest request = new UpdateRequest(indexName, typeName, docId);
    request.doc(doc);

    //3.通过client修改文档
    UpdateResponse response = client.update(request,RequestOptions.DEFAULT);


    //4.返回结果
    System.out.println(response.getResult());//UPDATED
}
```

5.2.3 删除文档

```
@Test
public void testDelete() throws IOException {
    //1.准备id数据
    String docId = "1001";

    //2.准备一个request对象
    DeleteRequest request = new DeleteRequest(indexName, typeName, docId);

    //3.通过client修改文档
    DeleteResponse response = client.delete(request, RequestOptions.DEFAULT);

    //4.返回结果
    System.out.println(response.getResult());//DELETED
}
```

5.3 批量操作文档

5.3.1 批量添加文档数据

```
@Test
public void testBulkAdd() throws IOException {
    //1.准备数据
    Novel n1 = new Novel(1001, "金瓶梅", "杨光", 1000L, new Date());
    Novel n2 = new Novel(1002, "聊斋", "蒲松龄", 2000L, new Date());
    Novel n3 = new Novel(1003, "红楼梦", "曹雪芹", 3000L, new Date());
    Novel n4 = new Novel(1004, "鬼吹灯", "浩男仔", 4000L, new Date());

    List<Novel> list = new ArrayList<>();
    list.add(n1);
    list.add(n2);
    list.add(n3);
    list.add(n4);


    ObjectMapper objectMapper = new ObjectMapper();
    //2.创建request
    BulkRequest bulkRequest = new BulkRequest();
    for (Novel n : list) {
        bulkRequest.add(new IndexRequest(indexName, typeName,
n.getId().toString()).source(objectMapper.writeValueAsString(n),
XContentType.JSON));
    }

    //3.client执行
    BulkResponse response = client.bulk(bulkRequest, RequestOptions.DEFAULT);


    //4.返回结果
    System.out.println(response);
}
```

5.3.2 批量删除

```
@Test
public void testBulkAdd() throws IOException {
    //1.准备数据
    List<String> list = new ArrayList<>();
    list.add("1001");
    list.add("1002");
    list.add("1003");

    //2.创建request
    BulkRequest bulkRequest = new BulkRequest();
    for (Novel n : list) {
        bulkRequest.add(new DeleteRequest(indexName, typeName,
n.getId().toString()));
    }

    //3.client执行
    BulkResponse response = client.bulk(bulkRequest, RequestOptions.DEFAULT);


    //4.返回结果
    System.out.println(response);
}
```

5.4 作业

> 创建索引，指定对应的数据结构
>
> 索引名：ssm-logs-index
>
> 类型名：ssm-logs-type
>
> 数据结构：

字段名	说明
createDate	创建时间
sendDate	发送时间
longCode	发送的长号码
mobile	手机号
corpName	发送的公司名称，需要分词检索
smsContent	下发的短信内容，需要分词检索
state	短信状态 0成功 1 失败
operatorId	运营商编号 1 移动 2 联通 3电信
province	省份
ipAddr	下发的服务器Ip地址
replyTotal	短信状态报告返回时长
fee	扣费

要求：

1.kibana中通过json实现

2.在java中通过RestClient实现

6、ES的各种查询API

6.1 term&terms查询

6.1.1 term查询

term查询，完成匹配查询，搜索前不会对你搜索的关键词进行分词，拿关键词去索引库进行匹配

```
GET /sms-logs-index/sms-logs-type/_search
{
  "from": 0, #开始的文档位置
  "size": 2, #显示文档的个数
  "query": {
    "term": {
      "province": {
        "value": "武汉"
      }
    }
  }
}
```

代码实现

```
@Test
public void testO1() throws IOException {
    //1.创建request对象
    SearchRequest request = new SearchRequest(indexName);
    request.types(typeName);


    //2.指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    //builder.from(0);
    // builder.size(2);
    builder.query(QueryBuilders.termQuery("province", "武汉"));
    request.source(builder);

    //3.执行查询
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);

    //4.显示查询结果
    SearchHit[] hits = response.getHits().getHits();
    if (hits != null && hits.length > 0) {
        for (SearchHit hit : hits) {
            Map<String,Object> map = hit.getSourceAsMap();
            System.out.println(map);
        }
    }

    client.close();
}
```

6.1.2 terms查询

> terms查询和term查询机制一样，都不会对搜索的关键词进行分词，直接去分词库进行匹配，找到就显示相应内容!
>
> terms针对于一个字段包含有多个值的情况
>
> term: province="北京"
>
> terms:province="北京" or province="上海" or province="武汉"

```
#terms多条件查询
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
      "terms": {
        "province": [
          "北京",
          "上海"
        ]
      }
    }
}
```

> 代码实现

```
@Test
```

```
public void testO2() throws IOException {
    //1.创建request对象
    SearchRequest request = new SearchRequest(indexName);
    request.types(typeName);


    //2.指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    //builder.from(0);
    // builder.size(2);
    builder.query(QueryBuilders.termsQuery("province","上海","北京"));//province=
上海 or province=北京
    request.source(builder);

    //3.执行查询
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);

    //4.显示查询结果
    SearchHit[] hits = response.getHits().getHits();
    if (hits != null && hits.length > 0) {
        for (SearchHit hit : hits) {
            Map<String,Object> map = hit.getSourceAsMap();
            System.out.println(map);
        }
    }

    client.close();
}
```

6.2 match查询

> 会自动根据类型不同采用相应的查询方式进行查询
>
> - 查询日期或数字类型，自动将字符串转换成对应的日期或这数字类型
> - 如果查询字段不能进行分词，match不会将关键词进行分词（smsContent="我是中国人”）
> - 如果查询字段是可以进行分词，match会自定将关键词进行分词，然后去分词库进行查询（smsContent="我" or smsContent= “中国人” or smsContent ="国人"）
>
> 其实match查询顶层还是基于term进行查询的，将多个term查询的结果封装在一起了

6.2.1 match_all查询

> 查询所有，不带任何条件

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "match_all": {} #查询所有文档数据
  }
}
```

> 代码实现

```
@Test
public void testO1() throws IOException {
    //1.创建request对象
    SearchRequest request = new SearchRequest(indexName);
```

```
    request.types(typeName);

    //2.指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    builder.from(0);
    builder.size(20);
    builder.query(QueryBuilders.matchAllQuery());
    request.source(builder);

    //3.执行查询
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);

    //4.显示查询结果
    SearchHit[] hits = response.getHits().getHits();
    if (hits != null && hits.length > 0) {
        for(SearchHit hit : hits){
            System.out.println(hit.getId());
        }
        System.out.println("总记录个数："+hits.length);
    }
}
```

6.2.2 match查询

> 指定一个Field作为查询条件

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "match": {
      "smsContent": "张三先生"  #因为smsContent的类型是text,则搜索时会自动将该关键词进行分词【张三，  三，先生】, smsContent="张三" or smsContent="三" or smsContent="先生"
    }
  }
}
```

```
@Test
public void test02() throws IOException {
    //1.创建request对象
    SearchRequest request = new SearchRequest(indexName);
    request.types(typeName);

    //2.指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    builder.from(0);
    builder.size(20);
    builder.query(QueryBuilders.matchQuery("smsContent", "张三先生"));
    request.source(builder);

    //3.执行查询
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);

    //4.显示查询结果
    SearchHit[] hits = response.getHits().getHits();
    if (hits != null && hits.length > 0) {
        for(SearchHit hit : hits){
            System.out.println(hit.getSourceAsMap().get("smsContent"));
```

```
            System.out.println();
        }
        System.out.println("总记录个数："+hits.length);
    }
}
```

6.2.3 布尔match查询

> 基于一个Field字段配置的内容进行分词查询，而且可以设置多个条件之间的关系：`and` 或者 `or`

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "match": {
      "smsContent": {
        "query":"中国 健康",
        "operator": "and" #内容中既有中国又要有健康
      }
    }
  }
}

GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "match": {
      "smsContent": {
        "query":"中国 健康",
        "operator": "or" #内容中包含中国或者健康
      }
    }
  }
}
```

> 代码实现

```
@Test
public void test03() throws IOException {
    //1.创建request对象
    SearchRequest request = new SearchRequest(indexName);
    request.types(typeName);

    //2.指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    builder.from(0);
    builder.size(20);
    builder.query(QueryBuilders.matchQuery("smsContent", "健康 中
国").operator(Operator.AND));
    request.source(builder);

    //3.执行查询
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);

    //4.显示查询结果
    SearchHit[] hits = response.getHits().getHits();
    if (hits != null && hits.length > 0) {
        for(SearchHit hit : hits){
```

```
            System.out.println(hit.getSourceAsMap().get("smsContent"));
            System.out.println();
        }
        System.out.println("总记录个数："+hits.length);
    }
}
```

6.2.4 multi_match查询

> match针对于一个字段进行匹配查询，而multi_match可以针对于多个字段进行查询，而且多个字段对应一个text

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "multi_match": {
      "query": "武汉", #指定搜索的关键词
      "fields": ["province","smsContent"]  #指定需要搜索的字段
     }
  }
}
```

> 代码实现

```
public static void queryTemplate(String indexName, String
typeName,RestHighLevelClient client, QueryBuilder qb) throws IOException {
    //1.创建request对象
    SearchRequest request = new SearchRequest(indexName);
    request.types(typeName);

    //2.指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    builder.from(0);
    builder.size(20);
    builder.query(qb);
    request.source(builder);

    //3.执行查询
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);

    //4.显示查询结果
    SearchHit[] hits = response.getHits().getHits();
    if (hits != null && hits.length > 0) {
        for(SearchHit hit : hits){
            System.out.println(hit.getSourceAsMap().get("smsContent"));
            System.out.println();
        }
        System.out.println("总记录个数："+hits.length);
    }
}

public static void main(String[] args) throws IOException {
    QueryBuilder qb = QueryBuilders.multiMatchQuery("北京","province",
"smsContent");
    queryTemplate(indexName, typeName, client, qb);
```

```
}
```

6.3 其他查询

6.3.1 id查询

```
# where id=?
GET /sms-logs-index/sms-logs-type/256
```

> 代码实现

```
@Test
public void test01() throws IOException {
    //1.构成request对象
    GetRequest request = new GetRequest(indexName, typeName, "21");

    //2.执行查询
    GetResponse response = client.get(request, RequestOptions.DEFAULT);

    //3.输出结果
    System.out.println(response.getSourceAsMap());
}
```

6.3.2 ids查询

> where ids in (21,22, 23)

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "ids": {
      "values": ["20","21","22"]
    }
  }
}
```

> 代码实现

```
@Test
public void test02() throws IOException {
    IdsQueryBuilder qb = QueryBuilders.idsQuery().addIds("20", "21", "22");
    Match.queryTemplate(indexName, typeName, client, qb);
}
```

6.3.3 fuzzy查询

> 类似于模糊查询，输入的关键词和匹配的词允许存在偏差
>
> 查询关键词：appla--》apple applf appld

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "fuzzy": {
      "corpName": {
        "value": "盒马先生",
        "fuzziness": 1 #偏差的个数
      }
    }
  }
}
```

> 代码实现

```
@Test
public void test03() throws IOException {
    QueryBuilder qb = QueryBuilders.fuzzyQuery("corpName","盒马先
生").fuzziness(Fuzziness.ONE);
    Match.queryTemplate(indexName, typeName, client, qb);
}
```

6.3.4 wildcard查询

> 通配符查询，和MySQL中Like是一样的用法，可以指定*和?

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "wildcard": {
      "corpName": {
        "value": "*移动" #可以使用* 或者 ? 通配符
      }
    }
  }
}
```

> 代码实现

```
@Test
public void test04() throws IOException {
    QueryBuilder qb = QueryBuilders.wildcardQuery("corpName", "中国*");

    Match.queryTemplate(indexName, typeName, client, qb);
}
```

6.3.5 range查询

> 范围查询，只能针对数值类型，可以根据Field大小关系查询

```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
   "range": {
     "fee": {
       "gt": 5,
       "lt": 10 # 5<fee<10  gte >=   gt> lt <  lte <=
     }
   }
  }
}
```

> 代码实现

```
QueryBuilder qb = QueryBuilders.rangeQuery("fee").gt(5).lt(10);
。。。
```

6.3.6 regexp查询

> 正则查询，可以通过正则表达式进行查询

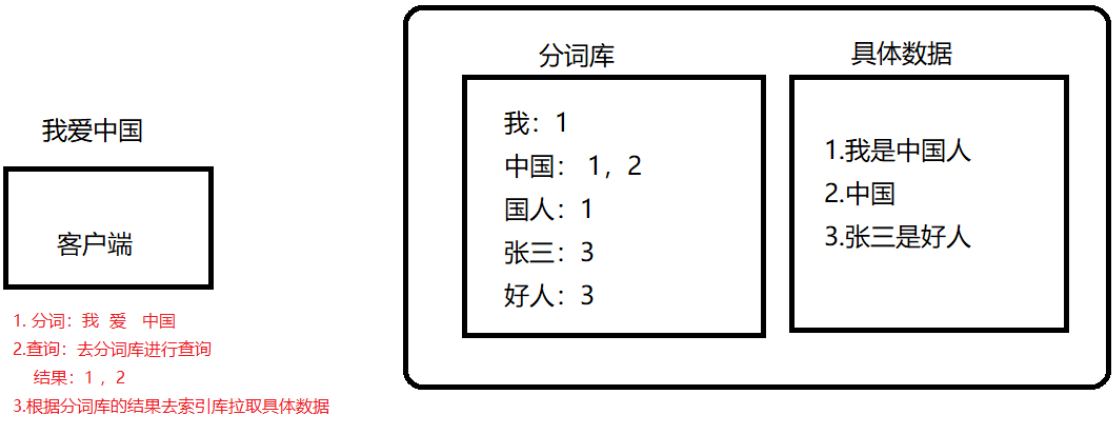
```
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
      "regexp": {
        "mobile": "139[0-9]{8}"#正则表达式 以139开头的11位手机号
      }
  }
}
```

> 代码实现

```
QueryBuilder qb = QueryBuilders.regexpQuery("mobile", "139[0-9]{8}");
```

6.4 Scroll分页

> 倒排索引算法
>
> - 将存放的数据按照一定的方式进行分词，然后将分词的内容存在一个分词库中。
> - 当用户查询数据时，会将用户的关键词进行分词
> - 然后去分词库中进行匹配，最终得到的数据的标识（id）
> - 根据这个标识id去索引库拉取真实数据



> 分页的方式主要有两种：
>
> - from+size:存在局限性，一般索引的数据量不能超过1W
>
> 1、先根据用户的关键词进行分词，
> 2、要去分词库进行检索，获取索引id
> 3、根据索引id拉取索引数据，这个过程非常耗时
> 4、根据score进行排序，这个过程也是非常耗时
> 5、根据这个from+size开始显示索引数据，然后舍弃一部分
> 6、返回结果
>
> - scroll+size
>
> 1、先根据用户的关键词进行分词，
> 2、要去分词库进行检索，获取文档id
> 3、会将获取的id存放在一个ES的上下文中
> 4、然后根据指定的size去ES的上下文环境中拉取数据，拉取完成后，再从ES上下文中移除
> 5、然后去索引库检索数据
> 6、如果进行下一页查询，直接去es上下文中进行检索，重复第4和第5步
>
> Scroll分页方式不适合做实时的数据分页查询

```
#根据size从ES上下文中获取数据，1m表示文档id在es上下文中的生存时间
GET /sms-logs-index/sms-logs-type/_search?scroll=1m
{
  "query": {
    "match_all": {}
  },
  "size":2
}

#需要根据scroll_id查询下一页
GET /_search/scroll
{
 "scroll_id":"DnF1ZXJ5VGhlbkZldGNoAwAAAAAAADjUFnRvTjJsSEo2UUdtLXd0RFpLckNxU1EAAAAAAAAA41RZObO4ybEhKNlFHbS13dERaS3JDcVNRAAAAAAAAAONYWdG9OMmxISjZRR2Otd3REWktyQ3FTUQ==" ,
  "scroll":"1m"
}

GET /_search/scroll
```

```
{
 "scroll_id":"根据第一步查询等到的scroll_id" ,
  "scroll":"指定es中id的存活时间"
}

#删除指定的es上下文中的数据
DELETE /_search/scroll/scroll_id带了吗
```

> 代码实现

```
// Java实现scroll分页
@Test
public void scrollQuery() throws IOException {
    //1. 创建SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定scroll信息
    request.scroll(TimeValue.timeValueMinutes(1L));

    //3. 指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    builder.size(4);
    builder.sort("fee", SortOrder.DESC);
    builder.query(QueryBuilders.matchAllQuery());

    request.source(builder);

    //4. 获取返回结果scrollId, source
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    String scrollId = resp.getScrollId();
    System.out.println("----------首页---------");
    for (SearchHit hit : resp.getHits().getHits()) {
        System.out.println(hit.getSourceAsMap());
    }


    while(true) {
        //5. 循环 - 创建SearchScrollRequest
        SearchScrollRequest scrollRequest = new SearchScrollRequest(scrollId);

        //6. 指定scrollId的生存时间
        scrollRequest.scroll(TimeValue.timeValueMinutes(1L));

        //7. 执行查询获取返回结果
        SearchResponse scrollResp = client.scroll(scrollRequest,
RequestOptions.DEFAULT);

        //8. 判断是否查询到了数据，输出
        SearchHit[] hits = scrollResp.getHits().getHits();
        if(hits != null && hits.length > 0) {
            System.out.println("----------下一页---------");
            for (SearchHit hit : hits) {
                System.out.println(hit.getSourceAsMap());
            }
        }else{
```

```
            //9. 判断没有查询到数据-退出循环
            System.out.println("----------结束---------");
            break;
        }
    }


    //10. 创建CLearScrollRequest
    ClearScrollRequest clearScrollRequest = new ClearScrollRequest();

    //11. 指定ScrollId
    clearScrollRequest.addScrollId(scrollId);

    //12. 删除ScrollId
    ClearScrollResponse clearScrollResponse =
client.clearScroll(clearScrollRequest, RequestOptions.DEFAULT);

    //13. 输出结果
    System.out.println("删除scroll: " + clearScrollResponse.isSucceeded());

}
```

6.5 delete-by-query

> 根据term，match等查询方式去删除大量的文档
>
> Ps：如果你需要删除的内容，是index下的大部分数据，推荐创建一个全新的index，将保留的文档内容，添加到全新的索引

```
# delete-by-query
POST /sms-logs-index/sms-logs-type/_delete_by_query
{
  "query": {
    "range": {
      "fee": {
        "lt": 4
      }
    }
  }
}
```

> 代码实现方式

```
// Java代码实现
@Test
public void deleteByQuery() throws IOException {
    //1. 创建DeleteByQueryRequest
    DeleteByQueryRequest request = new DeleteByQueryRequest(index);
    request.types(type);

    //2. 指定检索的条件     和SearchRequest指定Query的方式不一样
    request.setQuery(QueryBuilders.rangeQuery("fee").lt(4));

    //3. 执行删除
    BulkByScrollResponse resp = client.deleteByQuery(request,
RequestOptions.DEFAULT);
```

```
    //4. 输出返回结果
    System.out.println(resp.toString());

}
```

6.6 复合查询

6.6.1 bool查询

> 复合过滤器，将你的多个查询条件，以一定的逻辑组合在一起。
>
> - must： 所有的条件，用must组合在一起，表示And的意思
> - must_not：将must_not中的条件，全部都不能匹配，标识Not的意思
> - should：所有的条件，用should组合在一起，表示Or的意思

```
# 查询省份为武汉或者北京
# 运营商不是联通
# smsContent中包含中国和平安
# bool查询
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "term": {
            "province": {
              "value": "北京"
            }
          }
        },
        {
          "term": {
            "province": {
              "value": "武汉"
            }
          }
        }
      ],
      "must_not": [
        {
          "term": {
            "operatorId": {
              "value": "2"
            }
          }
        }
      ],
      "must": [
        {
          "match": {
            "smsContent": "中国"
          }
        },
        {
          "match": {
```

```
            "smsContent": "平安"
          }
        }
      ]
    }
  }
}
```

> 代码实现方式

```
// Java代码实现Bool查询
@Test
public void BoolQuery() throws IOException {
    //1. 创建SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
    // # 查询省份为武汉或者北京
    boolQuery.should(QueryBuilders.termQuery("province","武汉"));
    boolQuery.should(QueryBuilders.termQuery("province","北京"));
    // # 运营商不是联通
    boolQuery.mustNot(QueryBuilders.termQuery("operatorId",2));
    // # smsContent中包含中国和平安
    boolQuery.must(QueryBuilders.matchQuery("smsContent","中国"));
    boolQuery.must(QueryBuilders.matchQuery("smsContent","平安"));

    builder.query(boolQuery);
    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 输出结果
    for (SearchHit hit : resp.getHits().getHits()) {
        System.out.println(hit.getSourceAsMap());
    }
}
```

6.6.2 boosting查询

> boosting查询可以帮助我们去影响查询后的score。
>
> - positive：只有匹配上positive的查询的内容，才会被放到返回的结果集中。
> - negative：如果匹配上positive并且也匹配上了negative，就可以降低这样的文档score。
> - negative_boost：指定系数，必须小于1.0 ，那么匹配到的内容会将分数乘以当前系数
>
> 关于查询时，分数是如何计算的：
>
> - 搜索的关键字在文档中出现的频次越高，分数就越高
> - 指定的文档内容越短，分数就越高
> - 我们在搜索时，指定的关键字也会被分词，这个被分词的内容，被分词库匹配的个数越多，分数越高

```
# boosting查询  收货安装
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "boosting": {
      "positive": {
        "match": {
          "smsContent": "收货安装"
        }
      },
      "negative": {
        "match": {
          "smsContent": "王五"
        }
      },
      "negative_boost": 0.5
    }
  }
}
```

> 代码实现方式

```
// Java实现Boosting查询
@Test
public void BoostingQuery() throws IOException {
    //1. 创建SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    BoostingQueryBuilder boostingQuery = QueryBuilders.boostingQuery(
            QueryBuilders.matchQuery("smsContent", "收货安装"),
            QueryBuilders.matchQuery("smsContent", "王五")
    ).negativeBoost(0.5f);

    builder.query(boostingQuery);
    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 输出结果
    for (SearchHit hit : resp.getHits().getHits()) {
        System.out.println(hit.getSourceAsMap());
    }
}
```

6.7 filter查询

> query，根据你的查询条件，去计算文档的匹配度得到一个分数，并且根据分数进行排序，不会做缓存的。
>
> filter，根据你的查询条件去查询文档，不去计算分数，而且filter会对经常被过滤的数据进行缓存。

```
# filter查询
GET /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "term": {
            "corpName": "盒马鲜生"
          }
        },
        {
          "range": {
            "fee": {
              "lte": 4
            }
          }
        }
      ]
    }
  }
}
```

> 代码实现方式

```
// Java实现filter操作
@Test
public void filter() throws IOException {
    //1. SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 查询条件
    SearchSourceBuilder builder = new SearchSourceBuilder();
    BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
    boolQuery.filter(QueryBuilders.termQuery("corpName","盒马鲜生"));
    boolQuery.filter(QueryBuilders.rangeQuery("fee").lte(5));

    builder.query(boolQuery);
    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 输出结果
    for (SearchHit hit : resp.getHits().getHits()) {
        System.out.println(hit.getSourceAsMap());
    }


}
```

6.8 高亮查询【重点】

> 高亮查询就是你用户输入的关键字，以一定的特殊样式展示给用户，让用户知道为什么这个结果被检索出来。
>
> 高亮展示的数据，本身就是文档中的一个Field，单独将Field以highlight的形式返回给你。
>
> ES提供了一个highlight属性，和query同级别的。

- fragment_size：指定高亮数据展示多少个字符回来。
- pre_tags：指定前缀标签，举个栗子< font color="red" >
- post_tags：指定后缀标签，举个栗子< /font >
- fields：指定哪几个Field以高亮形式返回

效果图

> RESTful实现

```
# highlight查询
POST /sms-logs-index/sms-logs-type/_search
{
  "query": {
    "match": {
      "smsContent": "盒马"
    }
  },
```

```
  "highlight": {
    "fields": {
      "smsContent": {}
    },
    "pre_tags": "<font color='red'>",
    "post_tags": "</font>",
    "fragment_size": 10
  }
}
```

> 代码实现方式

```
// Java实现高亮查询
@Test
public void highLightQuery() throws IOException {
    //1. SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定查询条件（高亮）
    SearchSourceBuilder builder = new SearchSourceBuilder();
    //2.1 指定查询条件
    builder.query(QueryBuilders.matchQuery("smsContent","盒马"));
    //2.2 指定高亮
    HighlightBuilder highlightBuilder = new HighlightBuilder();
    highlightBuilder.field("smsContent",10)
            .preTags("<font color='red'>")
            .postTags("</font>");
    builder.highlighter(highlightBuilder);

    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 获取高亮数据，输出
    for (SearchHit hit : resp.getHits().getHits()) {
        System.out.println(hit.getHighlightFields().get("smsContent"));
    }
}
```

6.9 聚合查询【重点】

> ES的聚合查询和MySQL的聚合查询类似，ES的聚合查询相比MySQL要强大的多，ES提供的统计数据的方式多种多样。

```
# ES聚合查询的RESTful语法
GET /index/type/_search
{
    "aggs": {
        "名字（agg）": {
            "agg_type": {
                "属性": "值"
            }
        }
    }
}
```

6.9.1 去重计数查询

> 去重计数，即Cardinality，第一步先将返回的文档中的一个指定的field进行去重，统计一共有多少条

```
# 去重计数查询 北京 上海 武汉 山西
GET /sms-logs-index/sms-logs-type/_search
{
  "aggs": {
    "agg": {
      "cardinality": {
        "field": "province"
      }
    }
  }
}
```

> 代码实现方式

```
//  Java代码实现去重计数查询
@Test
public void cardinality() throws IOException {
    //1. 创建SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定使用的聚合查询方式
    SearchSourceBuilder builder = new SearchSourceBuilder();

 builder.aggregation(AggregationBuilders.cardinality("agg").field("province"));

    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 获取返回结果
    Cardinality agg = resp.getAggregations().get("agg");
    long value = agg.getValue();
    System.out.println(value);
}
```

6.9.2 范围统计

> 统计一定范围内出现的文档个数，比如，针对某一个Field的值在 0~100,100~200,200~300之间文档出现的个数分别是多少。
>
> 范围统计可以针对普通的数值，针对时间类型，针对ip类型都可以做相应的统计。
>
> range，date_range，ip_range

> 数值统计

```
# 数值方式范围统计
GET /sms-logs-index/sms-logs-type/_search
{
  "aggs": {
    "agg": {
      "range": {
        "field": "fee",
        "ranges": [
          {
            "to": 5
          },
          {
            "from": 5,     # from有包含当前值的意思
            "to": 10
          },
          {
            "from": 10
          }
        ]
      }
    }
  }
}
```

> 时间范围统计

```
# 时间方式范围统计
POST /sms-logs-index/sms-logs-type/_search
{
  "aggs": {
    "agg": {
      "date_range": {
        "field": "createDate",
        "format": "yyyy",
        "ranges": [
          {
            "to": 2000
          },
          {
            "from": 2000
          }
        ]
      }
    }
  }
```

```
}
```

> ip统计方式

```
# ip方式 范围统计
POST /sms-logs-index/sms-logs-type/_search
{
  "aggs": {
    "agg": {
      "ip_range": {
        "field": "ipAddr",
        "ranges": [
          {
            "to": "10.126.2.9"
          },
          {
            "from": "10.126.2.9"
          }
        ]
      }
    }
  }
}
```

> 代码实现方式

```
// Java实现数值 范围统计
@Test
public void range() throws IOException {
    //1. 创建SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定使用的聚合查询方式
    SearchSourceBuilder builder = new SearchSourceBuilder();
    //-----------------------------------------------
    builder.aggregation(AggregationBuilders.range("agg").field("fee")
                                           .addUnboundedTo(5)
                                           .addRange(5,10)
                                           .addUnboundedFrom(10));
    //-----------------------------------------------
    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 获取返回结果
    Range agg = resp.getAggregations().get("agg");
    for (Range.Bucket bucket : agg.getBuckets()) {
        String key = bucket.getKeyAsString();
        Object from = bucket.getFrom();
        Object to = bucket.getTo();
        long docCount = bucket.getDocCount();
        System.out.println(String.format("key：%s，from：%s，to：%s，docCount：
%s",key,from,to,docCount));
    }
```

```
}
```

6.9.3 统计聚合查询

> 他可以帮你查询指定Field的最大值，最小值，平均值，平方和等
>
> 使用：extended_stats

```
# 统计聚合查询
POST /sms-logs-index/sms-logs-type/_search
{
  "aggs": {
    "agg": {
      "extended_stats": {
        "field": "fee"
      }
    }
  }
}
```

> 代码实现方式

```
// Java实现统计聚合查询
@Test
public void extendedStats() throws IOException {
    //1. 创建SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定使用的聚合查询方式
    SearchSourceBuilder builder = new SearchSourceBuilder();
    //----------------------------------------------
    builder.aggregation(AggregationBuilders.extendedStats("agg").field("fee"));
    //----------------------------------------------
    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 获取返回结果
    ExtendedStats agg = resp.getAggregations().get("agg");
    double max = agg.getMax();
    double min = agg.getMin();
    System.out.println("fee的最大值为：" + max + "，最小值为：" + min);
}
```

> 其他的聚合查询方式查看官方文档：https://www.elastic.co/guide/en/elasticsearch/reference/6.5/index.html

6.10 地图经纬度搜索

> ES中提供了一个数据类型 geo_point，这个类型就是用来存储经纬度的。
>
> 创建一个带geo_point类型的索引，并添加测试数据

```
# 创建一个索引，指定一个name，locaiton
PUT /map
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1
  },
  "mappings": {
    "map": {
      "properties": {
        "name": {
          "type": "text"
        },
        "location": {
          "type": "geo_point"
        }
      }
    }
  }
}


# 添加测试数据
PUT /map/map/1
{
  "name": "天安门",
  "location": {
    "lon": 116.403981,
    "lat": 39.914492
  }
}


PUT /map/map/2
{
  "name": "海淀公园",
  "location": {
    "lon": 116.302509,
    "lat": 39.991152
  }
}

PUT /map/map/3
{
  "name": "北京动物园",
  "location": {
    "lon": 116.343184,
    "lat": 39.947468
  }
}
```

6.10.1 ES的地图检索方式

语法	说明
geo_distance	直线距离检索方式
geo_bounding_box	以两个点确定一个矩形，获取在矩形内的全部数据
geo_polygon	以多个点，确定一个多边形，获取多边形内的全部数据

6.10.2 基于RESTful实现地图检索

> geo_distance

```
# geo_distance
POST /map/map/_search
{
  "query": {
    "geo_distance": {
      "location": {              # 确定一个点
        "lon": 116.433733,
        "lat": 39.908404
      },
      "distance": 3000,           # 确定半径
      "distance_type": "arc"      # 指定形状为圆形
    }
  }
}
```

> geo_bounding_box

```
# geo_bounding_box
GET /map/map/_search
{
  "query": {
    "geo_bounding_box": {
      "location": {
        "top_left": {               # 左上角的坐标点
          "lon": 116.326943,
          "lat": 39.95499
        },
        "bottom_right": {            # 右下角的坐标点
          "lon": 116.433446,
          "lat": 39.908737
        }
      }
    }
  }
}
```

> geo_polygon

```
# geo_polygon
GET /map/map/_search
{
  "query": {
    "geo_polygon": {
      "location": {
        "points": [                 # 指定多个点确定一个多边形
          {
            "lon": 116.298916,
            "lat": 39.99878
          },
          {
            "lon": 116.29561,
            "lat": 39.972576
          },
          {
            "lon": 116.327661,
            "lat": 39.984739
          }
        ]
      }
    }
  }
}
```

6.10.3 Java实现geo_polygon

```
// 基于Java实现geo_polygon查询
@Test
public void geoPolygon() throws IOException {
    //1. SearchRequest
    SearchRequest request = new SearchRequest(index);
    request.types(type);

    //2. 指定检索方式
    SearchSourceBuilder builder = new SearchSourceBuilder();
    List<GeoPoint> points = new ArrayList<>();
    points.add(new GeoPoint(39.99878,116.298916));
    points.add(new GeoPoint(39.972576,116.29561));
    points.add(new GeoPoint(39.984739,116.327661));
    builder.query(QueryBuilders.geoPolygonQuery("location",points));

    request.source(builder);

    //3. 执行查询
    SearchResponse resp = client.search(request, RequestOptions.DEFAULT);

    //4. 输出结果
    for (SearchHit hit : resp.getHits().getHits()) {
        System.out.println(hit.getSourceAsMap());
    }
}
```