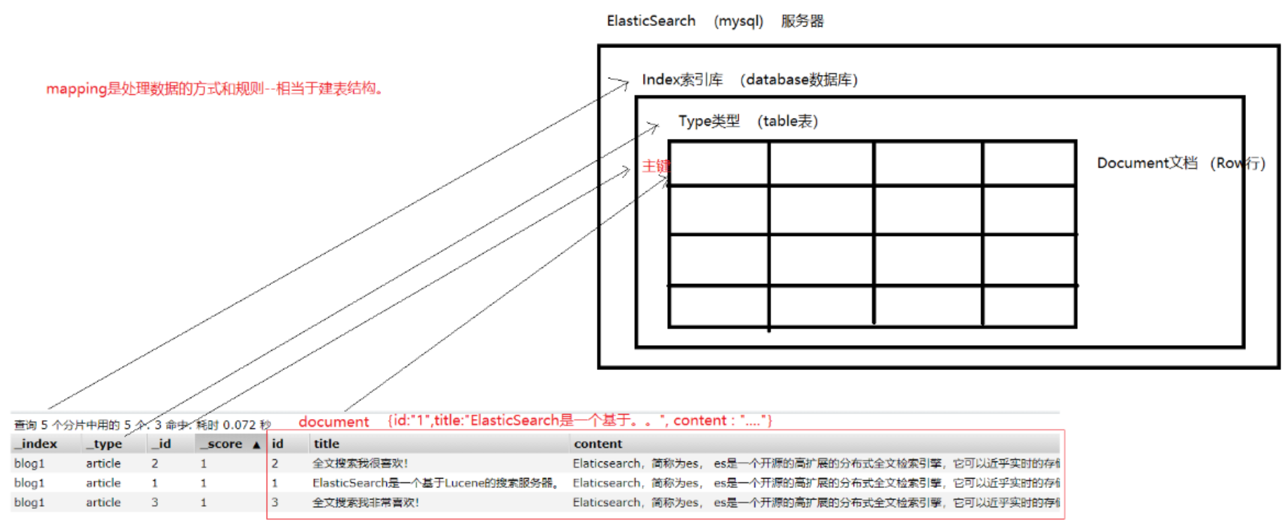
一.ElasticSearch相关概念

1.概述

ElasticSearch是面向文档的(document oriented)的,这意味这它可以存储整个或文档,然而它不仅是存储,它会索引每个文档的内容可以被搜索,在ElasticSearch中,可以对文档进行索引,搜索,排序,过滤.ES对比关系新数据库

```
Realctional DB -> Databases -> Tables -> Rows -> Columns
ElasticSearch -> Indices -> Types -> Documents -> Fields
```

2.核心概念



ElasticSearch (mysql) 服务器
Index索引库 (database数据库)
Type类型 (table表)
Document文档 (Row行)
主键
mapping是处理数据的方式和规则--相当于建表结构。
document {id:"1",title:"ElasticSearch是一个基于。。", content : "...."}

查询 5 个分片中用的 5 个. 3 命中. 耗时 0.072 秒

_index	_type	_id	_score ▲	id	title	content
blog1	article	2	1	2	全文搜索我很喜欢!	Elaticsearch，简称为es，es是一个开源的高扩展的分布式全文检索引擎，它可以近乎实时的存
blog1	article	1	1	1	ElasticSearch是一个基于Lucene的搜索服务器。	Elaticsearch，简称为es，es是一个开源的高扩展的分布式全文检索引擎，它可以近乎实时的存
blog1	article	3	1	3	全文搜索我非常喜欢!	Elaticsearch，简称为es，es是一个开源的高扩展的分布式全文检索引擎，它可以近乎实时的存

index就相当于数据库种的database,type就相当于数据库中的table,id就相当于数据库中记录的主键,一个document文档就相当于数据库中的一条记录

二.ElasticSearch的入门

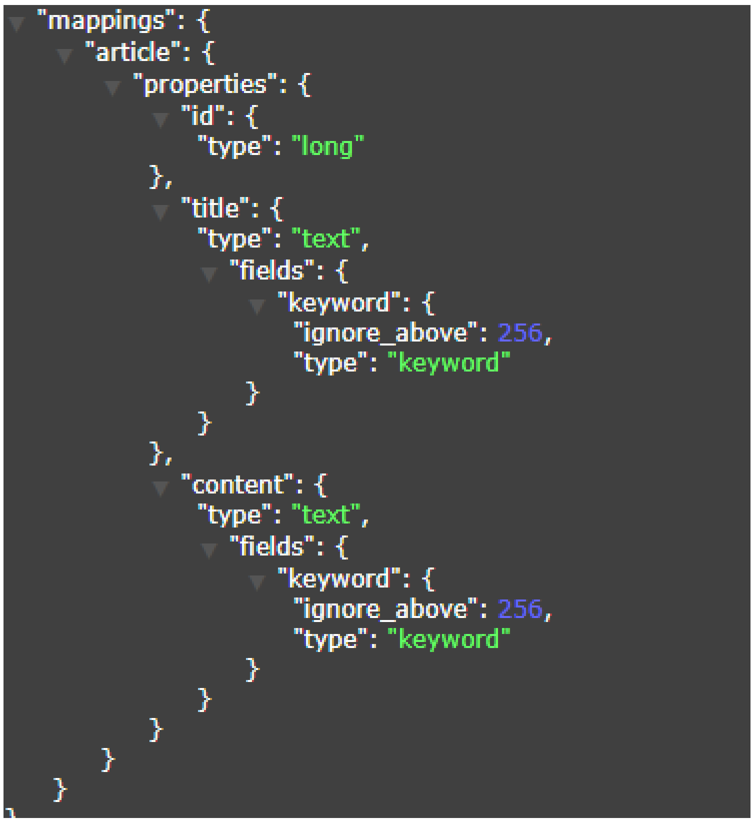
1.pom文件

```
<dependencies>
        <dependency>
            <groupId>org.elasticsearch</groupId>
            <artifactId>elasticsearch</artifactId>
            <version>5.6.8</version>
        </dependency>
        <dependency>
            <groupId>org.elasticsearch.client</groupId>
            <artifactId>transport</artifactId>
            <version>5.6.8</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-to-slf4j</artifactId>
            <version>2.9.1</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.24</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.21</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.10</version>
        </dependency>
    </dependencies>
```

2.新建索引

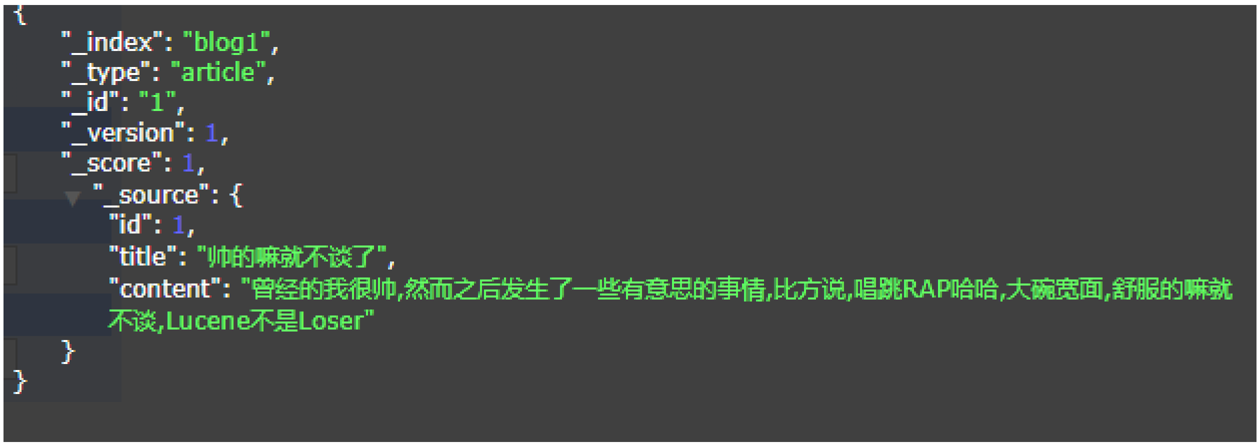
```
    @Test
    public void test01() throws IOException {
        //1.创建es客户端连接对象
        TransportClient transportClient = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.创建文档内容
        XContentBuilder builder = XContentFactory.jsonBuilder().
                startObject().field("id", 1)
                .field("title", "帅的嘛就不谈了")
                .field("content", "曾经的我很帅,然而之后发生了一些有意思的事情,比方说," +
                        "唱跳RAP哈哈,大碗宽面,舒服的嘛就不谈,Lucene不是Loser")
                .endObject();
        //3.建立文档对象
        transportClient.prepareIndex("blog1","article","1").setSource(builder).get();
        //4.释放资源
        transportClient.close();
    }
```

创建索引后,会自动生成Mapping映射:



```
"mappings": {
    "article": {
        "properties": {
            "id": {
              "type": "long"
            },
            "title": {
              "type": "text",
                "fields": {
                    "keyword": {
                      "ignore_above": 256,
                      "type": "keyword"
                    }
                }
            },
            "content": {
              "type": "text",
                "fields": {
                    "keyword": {
                      "ignore_above": 256,
                      "type": "keyword"
                    }
                }
            }
        }
    }
}
```

数据浏览:

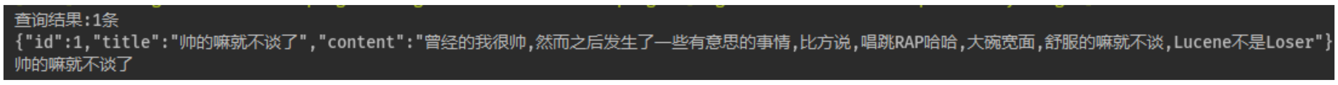


```
{
    "_index": "blog1",
    "_type": "article",
    "_id": "1",
    "_version": 1,
    "_score": 1,
    "_source": {
      "id": 1,
      "title": "帅的嘛就不谈了",
      "content": "曾经的我很帅,然而之后发生了一些有意思的事情,比方说,唱跳RAP哈哈,大碗宽面,舒服的嘛就不谈,Lucene不是Loser"
    }
}
```

3.全部查询

```
    /**
     * 查询全部
     * @throws Exception
     */
    @Test
    public void test02() throws Exception{
        //1.创建es客户端连接对象
        TransportClient transportClient = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.设置搜索条件
        SearchResponse searchResponse =
transportClient.prepareSearch("blog1").setTypes("article")
```

```
                .setQuery(QueryBuilders.matchAllQuery()).get();
        //3.遍历搜索结果数据
        SearchHits hits = searchResponse.getHits();//获取命中次数,查询结果有多少对象
        System.out.println("查询结果:"+hits.getTotalHits()+"条");
        Iterator<SearchHit> iterator = hits.iterator();
        while(iterator.hasNext()){
            SearchHit searchHit = iterator.next();
            System.out.println(searchHit.getSourceAsString());
            System.out.println(searchHit.getSource().get("title"));
        }
        transportClient.close();
    }
```



```
查询结果:1条
{"id":1,"title":"帅的嘛就不谈了","content":"曾经的我很帅,然而之后发生了一些有意思的事情,比方说,唱跳RAP哈哈,大碗宽面,舒服的嘛就不谈,Lucene不是Loser"}
帅的嘛就不谈了
```

4.字符串查询

```
@Test
    public void test05() throws Exception {
        //1、创建es客户端连接对象
        TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
                .addTransportAddress(new
                        InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"),
9300));
        //2、设置搜索条件
        SearchResponse searchResponse = client.prepareSearch("blog1")
                .setTypes("article")
                .setQuery(QueryBuilders.queryStringQuery("碗大")).get();
        //3、遍历搜索结果数据
        SearchHits hits = searchResponse.getHits(); // 获取命中次数，查询结果有多少对象
        System.out.println("查询结果有：" + hits.getTotalHits() + "条");
        Iterator<SearchHit> iterator = hits.iterator();
        while (iterator.hasNext()) {
            SearchHit searchHit = iterator.next(); // 每个查询对象
            System.out.println(searchHit.getSourceAsString()); // 获取字符串格式打印
            System.out.println("title:" + searchHit.getSource().get("title"));
        }
        //4、释放资源
        client.close();
    }
```

5. 词条查询

```
 @Test
    public void test03() throws Exception{
        //1.创建es客户端连接对象
        TransportClient transportClient = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
```

```
        //2.设置搜索条件
        SearchResponse searchResponse =
transportClient.prepareSearch("blog1").setTypes("article")
                .setQuery(QueryBuilders.termQuery("content","大碗")).get();

        //3.遍历搜索结果
        SearchHits hits = searchResponse.getHits();//获取命中次数,查询结果有多少对象
        System.out.println("查询结果:"+hits.getTotalHits()+"条");
        Iterator<SearchHit> iterator = hits.iterator();
        while(iterator.hasNext()){
            SearchHit searchHit = iterator.next();
            System.out.println(searchHit.getSourceAsString());
            System.out.println(searchHit.getSource().get("title"));
        }
        transportClient.close();
    }
```

ElasticSearch默认使用的分词器是将中文一个汉字一个汉字进行切割,所以我们使用termQuery进行词条查询,"大碗"就不会被搜寻到.如果单独的搜索"大","碗",就会搜寻到.

6. 模糊查询

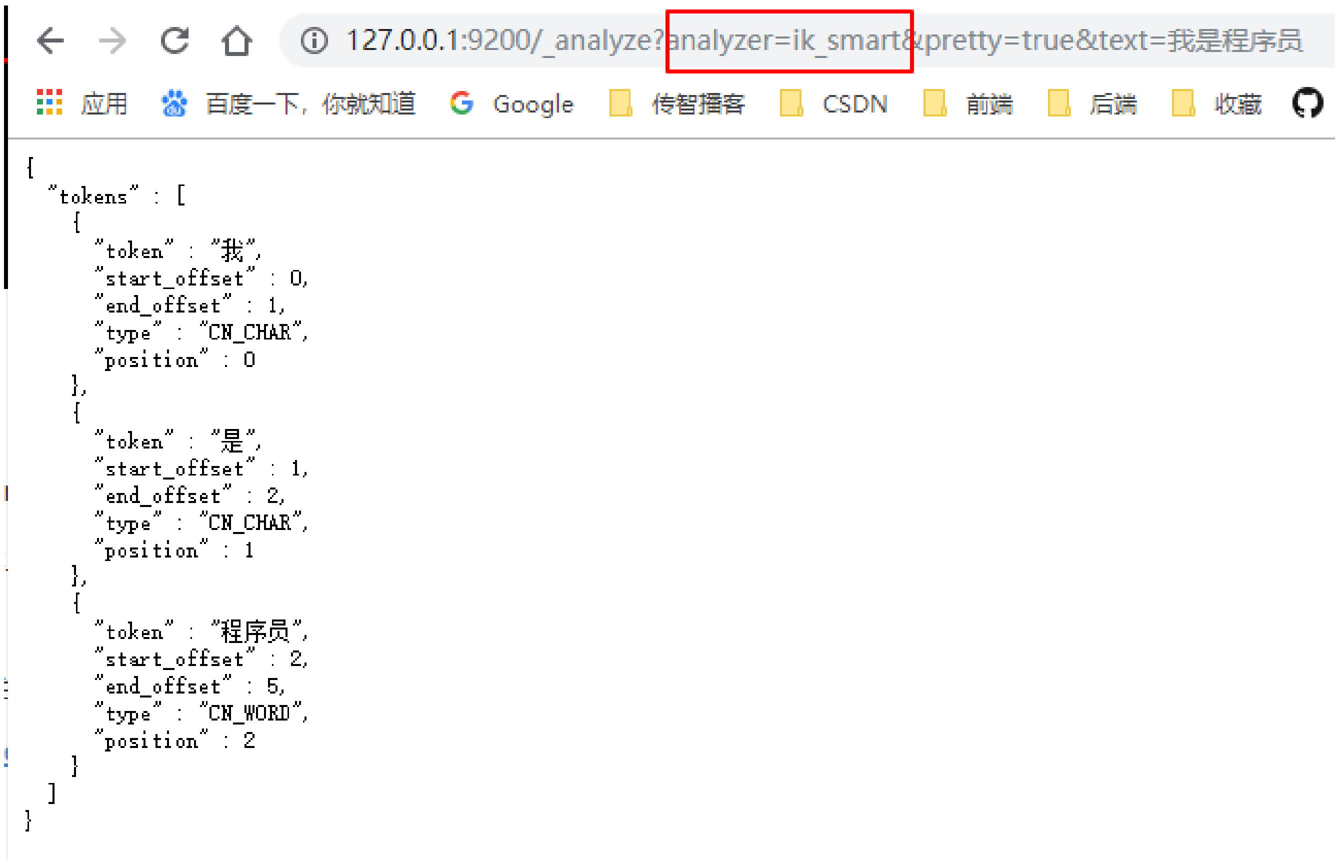
```
    @Test
    public void test04() throws Exception{
        //1.创建es客户端连接对象
        TransportClient transportClient = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));

        //2.设置搜索条件
        SearchResponse searchResponse =
transportClient.prepareSearch("blog1").setTypes("article")
                .setQuery(QueryBuilders.wildcardQuery("content","*大碗*")).get();

        //3.遍历搜索结果
        SearchHits hits = searchResponse.getHits();//获取命中次数,查询结果有多少对象
        System.out.println("查询结果:"+hits.getTotalHits()+"条");
        Iterator<SearchHit> iterator = hits.iterator();
        while(iterator.hasNext()){
            SearchHit searchHit = iterator.next();
            System.out.println(searchHit.getSourceAsString());
            System.out.println(searchHit.getSource().get("title"));
        }
        transportClient.close();
    }
```

模糊查询同样也是对词条进行模糊,如果使用默认分词,那么"大碗"模糊匹配的话,是不会匹配.

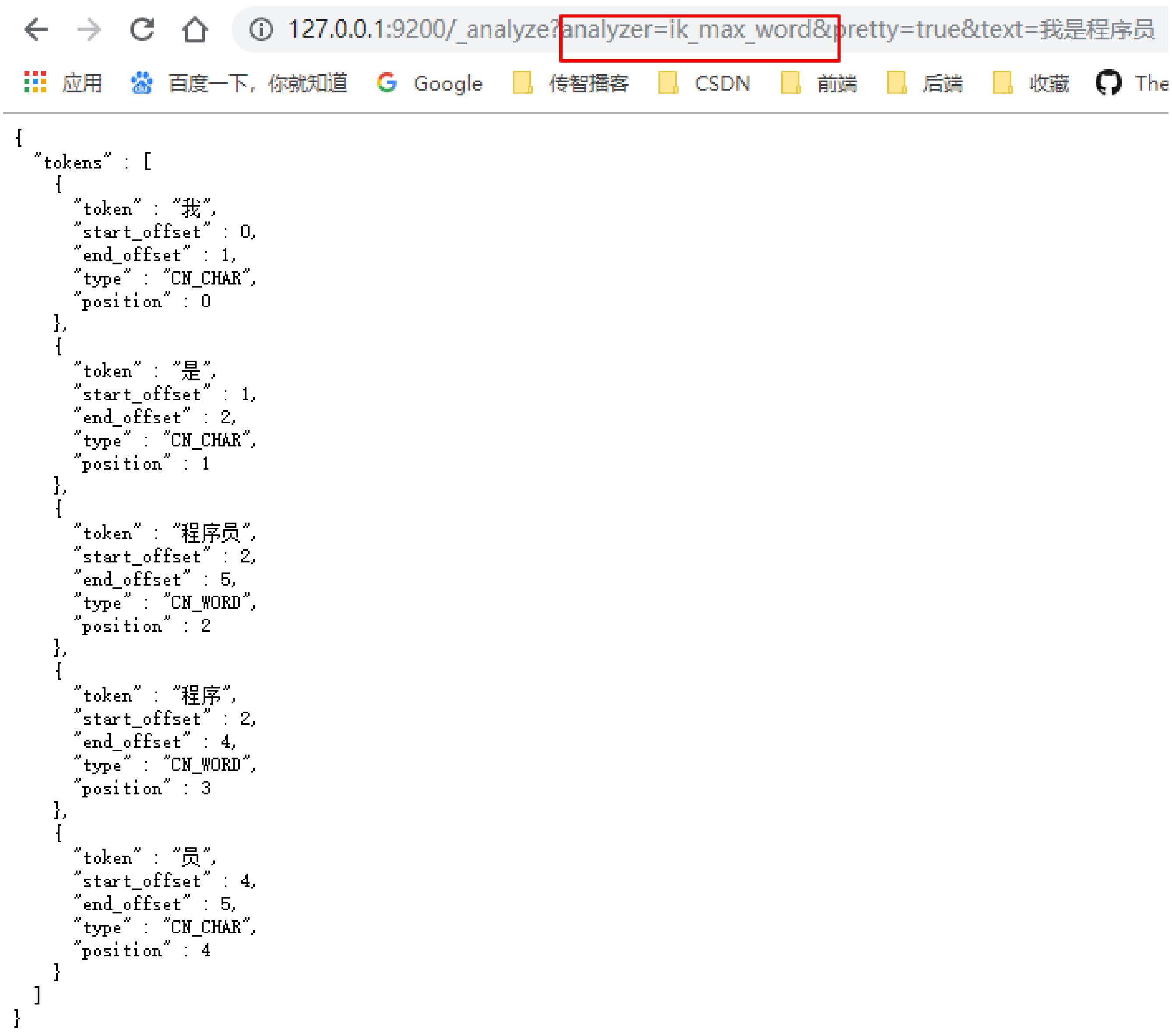
三.ik分词器

ik提供了两个分词算法ik_smart和ik_max_word,其中ik_smart为最少切分.ik_max_word为最细粒度划分.

我们可以在地址栏对ik分词器进行测试


127.0.0.1:9200/_analyze?analyzer=ik_smart&pretty=true&text=我是程序员
应用
百度一下，你就知道
Google
传智播客
CSDN
前端
后端
收藏

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "CN_CHAR",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "CN_CHAR",
      "position" : 1
    },
    {
      "token" : "程序员",
      "start_offset" : 2,
      "end_offset" : 5,
      "type" : "CN_WORD",
      "position" : 2
    }
  ]
}
```


127.0.0.1:9200/_analyze?analyzer=ik_max_word&pretty=true&text=我是程序员
应用
百度一下，你就知道
Google
传智播客
CSDN
前端
后端
收藏
The

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "CN_CHAR",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "CN_CHAR",
      "position" : 1
    },
    {
      "token" : "程序员",
      "start_offset" : 2,
      "end_offset" : 5,
      "type" : "CN_WORD",
      "position" : 2
    },
    {
      "token" : "程序",
      "start_offset" : 2,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 3
    },
    {
      "token" : "员",
      "start_offset" : 4,
      "end_offset" : 5,
      "type" : "CN_CHAR",
      "position" : 4
    }
  ]
}
```

四.索引操作

1.创建索引

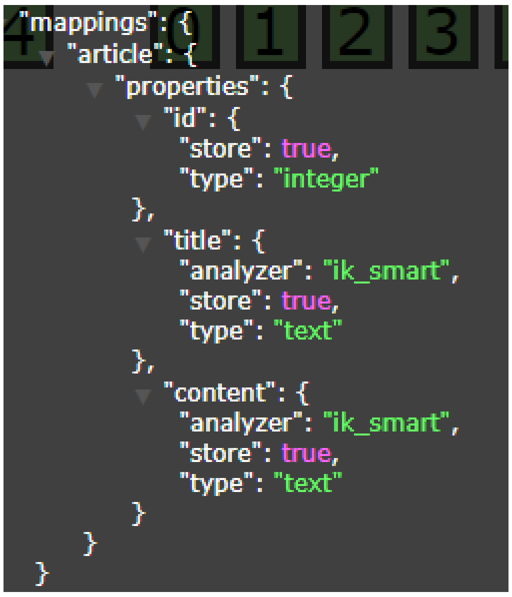
```
    @Test
    public void test11() throws Exception{
        //1.创建es客户端连接对象
        TransportClient transportClient = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.创建名称为blog2的索引
        transportClient.admin().indices().prepareCreate("blog2").get();
        //3.释放资源
        transportClient.close();
    }
```

2.删除索引

```
        @Test
    public void test12() throws Exception{
        //1.创建es客户端连接对象
        TransportClient transportClient = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.创建名称为blog2的索引
        transportClient.admin().indices().prepareDelete("blog2").get();
        //3.释放资源
        transportClient.close();
    }
```

五.映射操作

```
        @Test
    public void test13() throws Exception{
        //1.创建es客户端连接对象
        TransportClient transportClient = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.创建名称为blog2的索引
        transportClient.admin().indices().prepareCreate("blog2").get();
        //3.添加映射
        XContentBuilder builder = XContentFactory.jsonBuilder()
                .startObject()
                .startObject("article")
                .startObject("properties")
                .startObject("id")
                .field("type", "integer").field("store", "yes")
                .endObject()
                .startObject("title")
                .field("type", "string").field("store", "yes").field("analyzer", "ik_smart")
                .endObject()
                .startObject("content")
                .field("type", "string").field("store", "yes").field("analyzer", "ik_smart")
                .endObject()
                .endObject()
                .endObject()
                .endObject();
        //创建映射
        PutMappingRequest mapping = Requests.putMappingRequest("blog2")
                .type("article").source(builder);
        transportClient.admin().indices().putMapping(mapping).get();
        //释放资源
        transportClient.close();
    }
```


```
"mappings": {
  "article": {
    "properties": {
      "id": {
        "store": true,
        "type": "integer"
      },
      "title": {
        "analyzer": "ik_smart",
        "store": true,
        "type": "text"
      },
      "content": {
        "analyzer": "ik_smart",
        "store": true,
        "type": "text"
      }
    }
  }
```

总结:从"mappings"开始,json数据中一对大括号,就需要一对startObject( )和endObject( ),当中的属性适用field( )来进行链式编程.由于定义了analyzer:ik_smart,之后我们添加的文档elasticsearch就会对我们的文档进行ik分词.

六.文档操作

1.建立文档1

```
@Test
public void test14() throws Exception{
    //1.创建Client对象
    TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
            .addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));

    //2.创建文档信息
    XContentBuilder builder = XContentFactory.jsonBuilder()
            .startObject()
            .field("id", 1)
            .field("title", "ElasticSearch是一个基于Lucene的搜索服务器")
            .field("content", "它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口。
Elasticsearch是\n" +
                    "用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计
用于云计算中，能\n" +
                    "够达到实时搜索，稳定，可靠，快速，安装使用方便")
            .endObject();
    //3.建立文档对象
    client.prepareIndex("blog2","article","1").setSource(builder).get();
    //4.释放资源
    client.close();
}
```

这种建立文档的方式与我们的入门的是一样的.

2.建立文档2

在使用jackson转换实体的时候,我们需要导入依赖的坐标

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.8.1</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.8.1</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.8.1</version>
</dependency>
```

```
         @Test
    public void test15() throws Exception {
        //1.创建Client对象
        TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
                .addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));

        //2.描述json数据:
        Article article = new Article();
        article.setId(3);
        article.setTitle("玩蛇皮");
        article.setContent("嗯嗯,我是大碗宽,面,烨烨,你看这,我的面,它又大又宽");
        ObjectMapper objectMapper = new ObjectMapper();
        client.prepareIndex("blog2","article",article.getId().toString())
                .setSource(objectMapper.writeValueAsString(article)).get();
        //3.释放资源
        client.close();
    }
```

3.修改文档1

```
        @Test
    public void test16() throws Exception{
        //1.创建Client对象
        TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
                .addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.描述json数据
        Article article = new Article();
        article.setId(3);
        article.setTitle("玩蛇皮");
        article.setContent("嗯嗯,我是大碗宽,面,烨烨,你看这,我的面,他又小又瘦");
        ObjectMapper objectMapper = new ObjectMapper();
        client.prepareUpdate("blog2","article",article.getId().toString())
                .setDoc(objectMapper.writeValueAsString(article)).get();
        client.close();
    }
```

4.修改文档2

```
        @Test
    public void test17() throws Exception {
        //1.创建Client对象
        TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
                .addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.描述json数据
        Article article = new Article();
        article.setId(3);
        article.setTitle("玩蛇皮");
        article.setContent("嗯嗯,我是大碗宽,面,烨烨,你看这,我的面,他又小又瘦又宽又长哈哈");
        ObjectMapper objectMapper = new ObjectMapper();
        client.update(new
UpdateRequest("blog2","article",article.getId().toString()).doc(objectMapper.writeValueAsStr
ing(article))).get();
        client.close();
    }
```

5.删除文档

```
        @Test
    public void test18() throws Exception{
        //1.创建Client对象
        TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
                .addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.删除blog2下的类型是article中的id为1的数据
        client.delete(new DeleteRequest("blog2","article","1")).get();
        //3.释放资源
        client.close();

    }
```

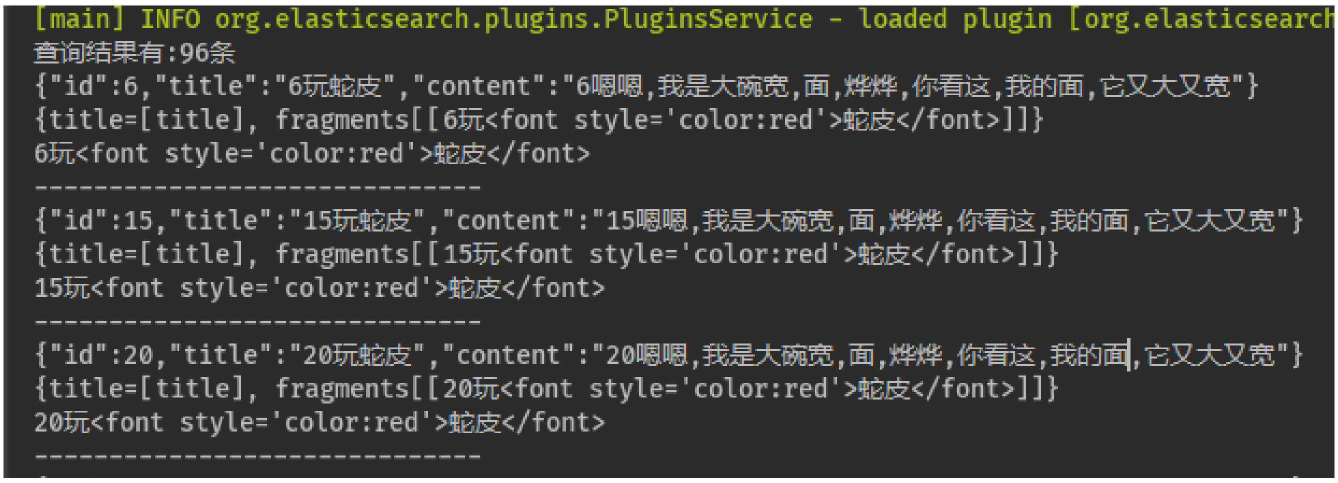
6.分页查询和排序

```
        @Test
    public void test20() throws Exception{
        //创建Client连接对象
        TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
```

```
                .addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName(
"127.0.0.1"), 9300));

        //搜索数据
        SearchRequestBuilder searchRequestBuilder =
                client.prepareSearch("blog2").setTypes("article")
                        .setQuery(QueryBuilders.matchAllQuery());

        //查询第2页数据,每页20条
        //setFrom();表示从第几条开始检索,默认是0
        //setSize();每页最多显示的记录数
        searchRequestBuilder.setFrom(20).setSize(20);
        //排序,根据id进行排序
        SearchResponse searchResponse = searchRequestBuilder.addSort("id",
SortOrder.DESC).get();

        SearchHits hits = searchResponse.getHits();
        System.out.println("查询结果有:"+hits.getTotalHits()+"条");
        Iterator<SearchHit> iterator = hits.iterator();
        while (iterator.hasNext()){
            SearchHit searchHit = iterator.next();//每个查询对象
            System.out.println(searchHit.getSourceAsString());//获取字符串格式打印
            System.out.println(searchHit.getSource().get("id"));
            System.out.println(searchHit.getSource().get("title"));
            System.out.println(searchHit.getSource().get("content"));
            System.out.println("-----------------------------");
        }
        client.close();
    }
```

7.查询结果高亮操作

```
        @Test
    public void test21() throws Exception{
        //1.创建Client连接对象
        TransportClient client = new PreBuiltTransportClient(Settings.EMPTY).
                addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        //2.搜索数据
        SearchRequestBuilder searchRequestBuilder =
client.prepareSearch("blog2").setTypes("article").
                setQuery(QueryBuilders.termQuery("title", "蛇皮"));
        //3.设置高亮数据
        HighlightBuilder highlightBuilder = new HighlightBuilder();
        highlightBuilder.preTags("<font style='color:red'>");
        highlightBuilder.postTags("</font>");
        highlightBuilder.field("title");
        searchRequestBuilder.highlighter(highlightBuilder);

        SearchResponse searchResponse = searchRequestBuilder.get();
        SearchHits hits = searchResponse.getHits();
        System.out.println("查询结果有:"+hits.getTotalHits()+"条");
        SearchHit[] hits1 = hits.getHits();
        for (SearchHit searchHit : hits1) {
            //String方式打印文档搜索内容
            System.out.println(searchHit.getSourceAsString());//获取字符串格式打印
            //打印高亮片段
            System.out.println(searchHit.getHighlightFields());
            //遍历高亮集合,打印高亮片段
            Text[] titles = searchHit.getHighlightFields().get("title").getFragments();
            for (Text title : titles) {
                System.out.println(title);
            }
            System.out.println("-------------------------------");
        }
        client.close();
    }
```



```
[main] INFO org.elasticsearch.plugins.PluginsService - loaded plugin [org.elasticsearch
查询结果有:96条
{"id":6,"title":"6玩蛇皮","content":"6嗯嗯,我是大碗宽,面,烨烨,你看这,我的面,它又大又宽"}
{title=[title], fragments[[6玩<font style='color:red'>蛇皮</font>]]}
6玩<font style='color:red'>蛇皮</font>
------------------------------
{"id":15,"title":"15玩蛇皮","content":"15嗯嗯,我是大碗宽,面,烨烨,你看这,我的面,它又大又宽"}
{title=[title], fragments[[15玩<font style='color:red'>蛇皮</font>]]}
15玩<font style='color:red'>蛇皮</font>
------------------------------
{"id":20,"title":"20玩蛇皮","content":"20嗯嗯,我是大碗宽,面,烨烨,你看这,我的面,它又大又宽"}
{title=[title], fragments[[20玩<font style='color:red'>蛇皮</font>]]}
20玩<font style='color:red'>蛇皮</font>
------------------------------
```

七.Spring Data ElasticSearch

1.编程实现

Spring Data是一个用于简化数据库访问,并支持云服务的开源框架,其主要目标是使得对数据的访问变得方便快捷,并支持map-reduce框架和云计算数据服务,Spring Data可以极大的简化JPA的写发,可以几乎不用写实现的情况下,实现对数据的访问和操作.除了CRUD外,还包括分页,排序等一些常用的功能

Spring Data ElasticSearch基于Spring Data API 简化elasticSearch操作,将原始操作elasticSearch的客户端API进行封装.Spring Data为ElasticSearch项目提供集成搜索引擎.Spring Data ElasticSearch的关键功能区域为中心的模型与ElasticSearch交互文档和轻松编写一个存储库数据访问层.

关键字	命名规则	解释	示例
and	findByField1AndField2	根据Field1和Field2获得数据	findByTitleAndContent
or	findByField1OrField2	根据Field1或Field2获得数据	findByTitleOrContent
is	findByField	根据Field获得数据	findByTitle
not	findByFieldNot	根据Field获得补集数据	findByTitleNot
between	findByFieldBetween	获得指定范围的数据	findByPriceBetween
lessThanEqual	findByFieldLessThan	获得小于等于指定值的数据	findByPriceLessThan

applicatiom.yml

```
spring:
  data:
    elasticsearch:
      cluster-Nodes: 127.0.0.1:9300
```

dao层:dao层我们需要继承ElasticsearchRepository接口.**如果想要根据条件查询，直接在方法名上拼接条件，如果需要分页，直接在参数上加上Pageable,如果我们需要排序，我们也可以在参数上拼接上Sort.**

```
public interface ArticleDao extends ElasticsearchRepository<Article,Integer> {
    Page<Article> findByTitleAndContent(String title, String content, Pageable pageable);
}
```

service实现层

```
@Service
public class ArticleServiceImpl implements ArticleService {
    @Autowired
    private ArticleDao articleDao ;

    public void save(Article article){
        articleDao.save(article);
    }

    @Override
    public Iterable<Article> findAll() {
        return articleDao.findAll();
    }

    @Override
    public Iterable<Article> findByPage(int page, int size) {
       return articleDao.findAll(PageRequest.of(page,size));
    }

    @Override
    public Page<Article> findBySortAndPage(PageRequest pageRequest) {
        Page<Article> page = articleDao.findAll(pageRequest);
        return page;
    }

    @Override
    public Page<Article> findBySortAndPageAndCondition(String title, String content,
Pageable pageable) {
        return articleDao.findByTitleAndContent(title,content,pageable);
    }

    @Override
    public Iterable<Article> findOnSortOnly(Sort sort) {
        return articleDao.findAll(sort);
    }
}
```

domain:

```
@Document(indexName = "fechin_blog01",type = "article")
indexName:索引名称(必填项)
type:索引类型

@Id:主键唯一标识

@Field(type = FieldType.Text,store = true,analyzer = "ik_smart",searchAnalyzer =
"ik_smart",index = true)
index:是否建立索引
analyzer:存储时使用的分词器
searchAnalyze:搜索时使用的分词器
store:是否存储
type:数据类型

注意:一旦添加了@Field注解,所有的默认值都不再生效,此外,如果添加了@Field注解,那么type字段必须指定.
```

```
@Document(indexName = "fechin_blog01",type = "article")
public class Article {
    @Field(index = false,type = FieldType.Integer,store = true)
    @Id
    private Integer id;
    @Field(type = FieldType.Text,store = true,analyzer = "ik_smart",searchAnalyzer =
"ik_smart",index = true)
    private String title;
    @Field(type = FieldType.Text,store = true,analyzer = "ik_smart",searchAnalyzer =
"ik_smart",index = true)
    private String content;

    @Override
    public String toString() {
        return "Article{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", content='" + content + '\'' +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

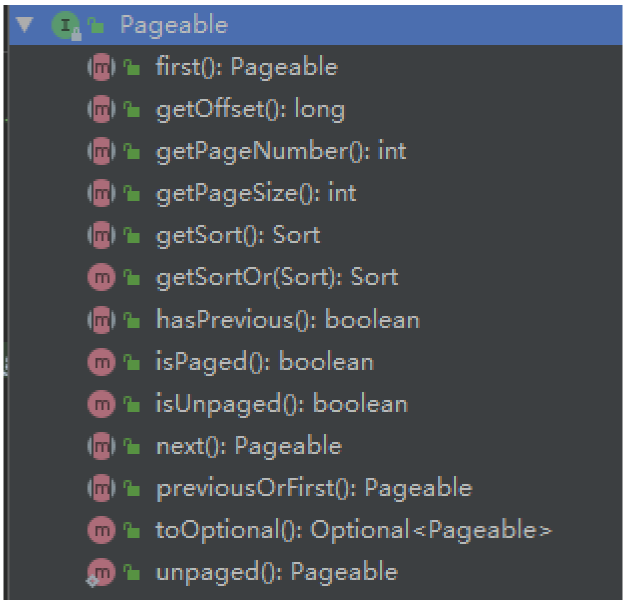
service接口层和controller层略

测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringbootEsTest01ApplicationTests {

    @Autowired
    private ArticleService articleService;

    @Test
    public void save(){
        Article article = new Article();
        article.setId(1);
        article.setTitle("搜索");
        article.setContent("你搜谁呢");
        articleService.save(article);
    }

    @Test
    public void save100(){
        for (int i = 0; i < 100; i++) {
            Article article = new Article();
            article.setId(i);
            article.setTitle(i+"搜索");
            article.setContent(i+"你搜谁呢,臭滴滴滴");
            articleService.save(article);
        }
    }

    @Test
    public void findAll(){
        Iterable<Article> all = articleService.findAll();
        for (Article article : all) {
            System.out.println(article.getContent());
        }
    }
        /**分页*/
    @Test
    public void findByPage(){
        Iterable<Article> page = articleService.findByPage(0,10);
        for (Article article : page) {
            System.out.println(article.getContent());
        }
    }
        /**分页排序*/
    @Test
    public void findBySortAndPage(){
        PageRequest request = PageRequest.of(0, 10, Sort.by(Sort.Order.asc("id")));
        Page<Article> page = articleService.findBySortAndPage(request);
        for (Article article : page) {
            String content = article.getContent();
            System.out.println(content);
        }
    }

    /**根据条件,分页排序*/
    @Test
    public void findBySortAndPageAndCondition(){
        PageRequest request = PageRequest.of(0, 10, Sort.by(Sort.Order.desc("id")));
        Page<Article> page = articleService.findBySortAndPageAndCondition("搜索", "搜",
request);
        for (Article article : page) {
            System.out.println(article);
        }
    }

    /**只进行排序*/
    @Test
    public void findBySortOnly(){
        Iterable<Article> sortOnly =
articleService.findOnSortOnly(Sort.by(Sort.Order.desc("id")));
        for (Article article : sortOnly) {
            System.out.println(article);
        }
    }
}
```

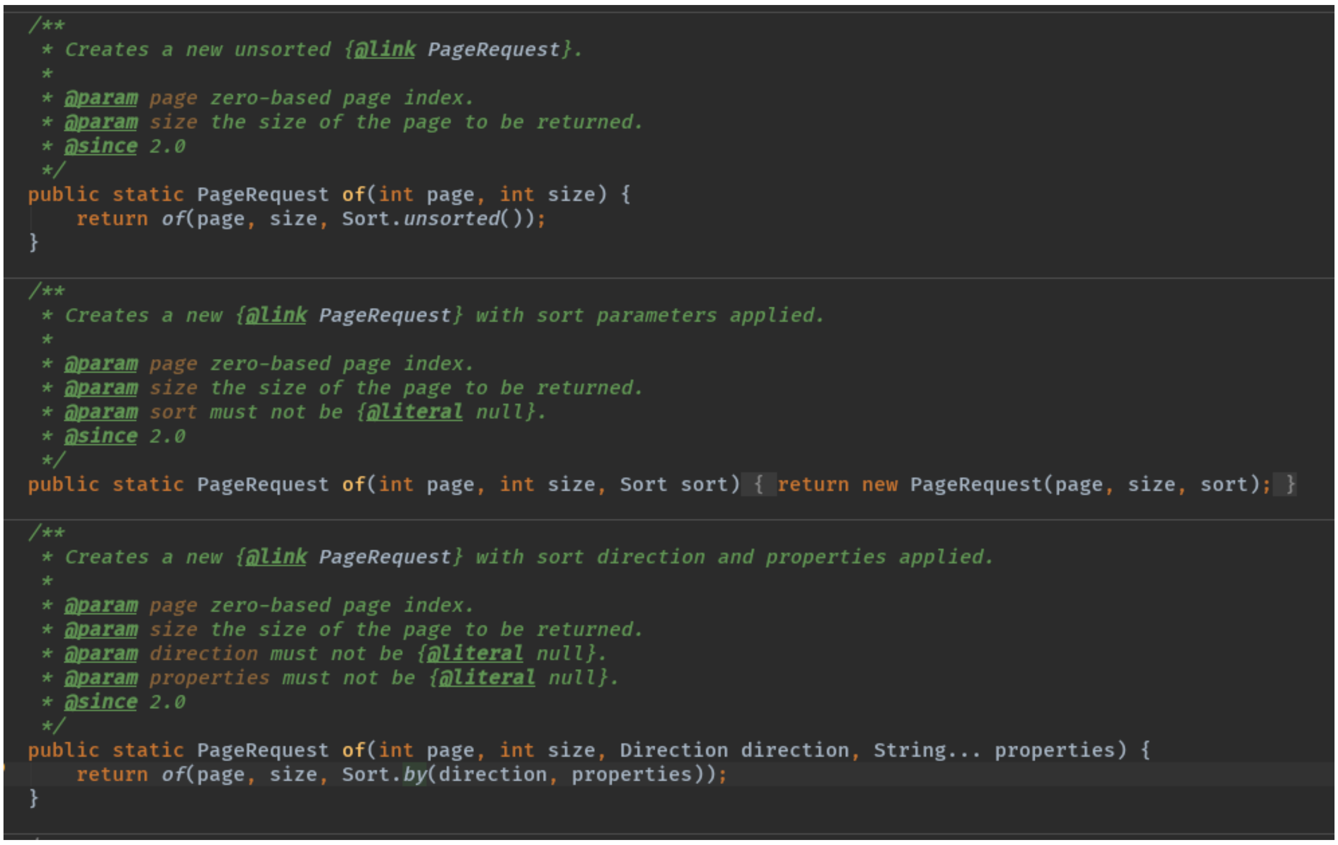
2.关于分页的Pageable接口分析

```
---Pageable
        ---AbstractPageRequest
                ---QPageRequest
                ---PageRequest
        ---Unpaged
```



```
Pageable
    first(): Pageable
    getOffset(): long
    getPageNumber(): int
    getPageSize(): int
    getSort(): Sort
    getSortOr(Sort): Sort
    hasPrevious(): boolean
    isPaged(): boolean
    isUnpaged(): boolean
    next(): Pageable
    previousOrFirst(): Pageable
    toOptional(): Optional<Pageable>
    unpaged(): Pageable
```

以上是Pageable接口

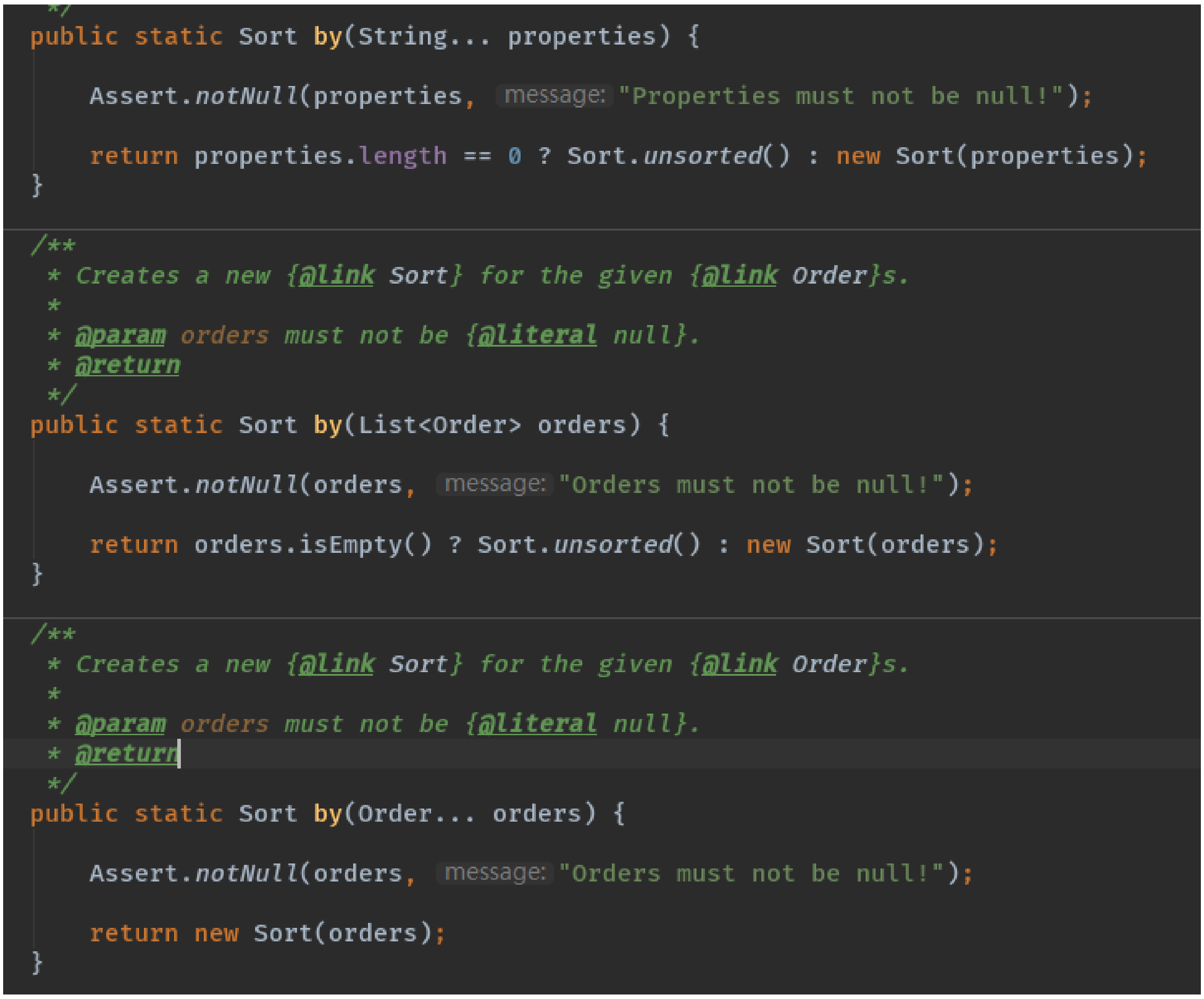


```java
/**
 * Creates a new unsorted {@link PageRequest}.
 *
 * @param page zero-based page index.
 * @param size the size of the page to be returned.
 * @since 2.0
 */
public static PageRequest of(int page, int size) {
    return of(page, size, Sort.unsorted());
}

/**
 * Creates a new {@link PageRequest} with sort parameters applied.
 *
 * @param page zero-based page index.
 * @param size the size of the page to be returned.
 * @param sort must not be {@literal null}.
 * @since 2.0
 */
public static PageRequest of(int page, int size, Sort sort) { return new PageRequest(page, size, sort); }

/**
 * Creates a new {@link PageRequest} with sort direction and properties applied.
 *
 * @param page zero-based page index.
 * @param size the size of the page to be returned.
 * @param direction must not be {@literal null}.
 * @param properties must not be {@literal null}.
 * @since 2.0
 */
public static PageRequest of(int page, int size, Direction direction, String... properties) {
    return of(page, size, Sort.by(direction, properties));
}
```

以上是PageRequest类,有很多分页方法都已经过时.前两种我们就可以完全够我们使用.

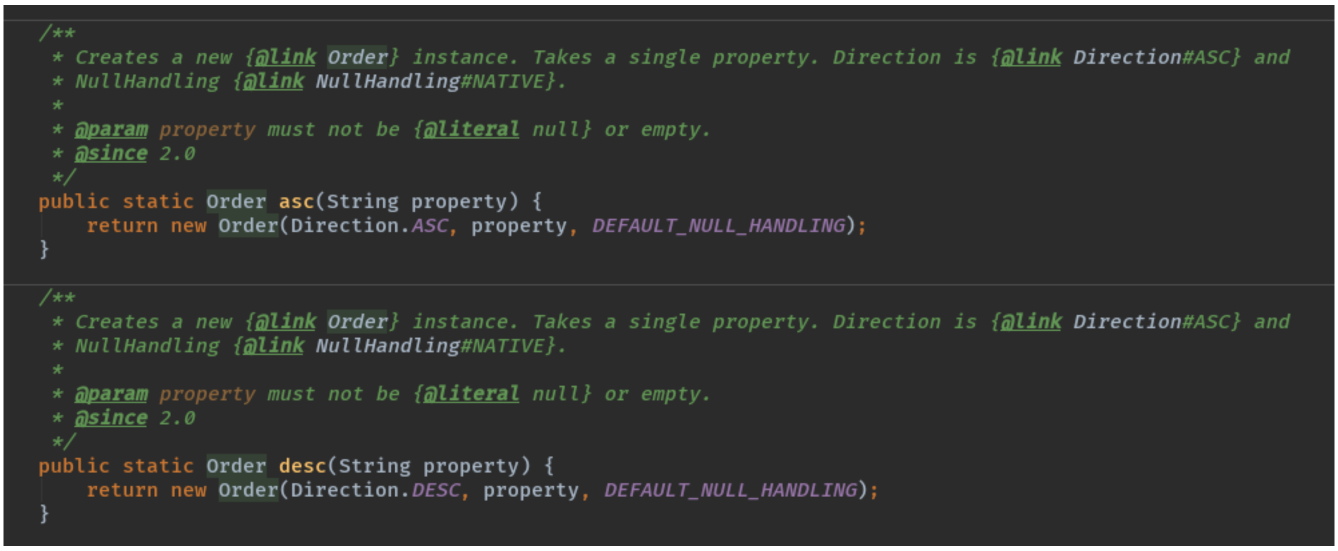
```
PageRequest.of(int page,int size);
PageRequest.of(int page,int size,Sort sort);
PageRequest.of(int page,int size,Direction direction,String...properties);
```

Sort是Spring给我们提供的类.



```
 */
public static Sort by(String... properties) {

    Assert.notNull(properties, message: "Properties must not be null!");

    return properties.length == 0 ? Sort.unsorted() : new Sort(properties);
}

/**
 * Creates a new {@link Sort} for the given {@link Order}s.
 *
 * @param orders must not be {@literal null}.
 * @return
 */
public static Sort by(List<Order> orders) {

    Assert.notNull(orders, message: "Orders must not be null!");

    return orders.isEmpty() ? Sort.unsorted() : new Sort(orders);
}

/**
 * Creates a new {@link Sort} for the given {@link Order}s.
 *
 * @param orders must not be {@literal null}.
 * @return
 */
public static Sort by(Order... orders) {

    Assert.notNull(orders, message: "Orders must not be null!");

    return new Sort(orders);
}
```

其中第三种最为常用,Order是Sort中的静态内部类.以下是Sort中最常用的两个方法



```
/**
 * Creates a new {@link Order} instance. Takes a single property. Direction is {@link Direction#ASC} and
 * NullHandling {@link NullHandling#NATIVE}.
 *
 * @param property must not be {@literal null} or empty.
 * @since 2.0
 */
public static Order asc(String property) {
    return new Order(Direction.ASC, property, DEFAULT_NULL_HANDLING);
}

/**
 * Creates a new {@link Order} instance. Takes a single property. Direction is {@link Direction#ASC} and
 * NullHandling {@link NullHandling#NATIVE}.
 *
 * @param property must not be {@literal null} or empty.
 * @since 2.0
 */
public static Order desc(String property) {
    return new Order(Direction.DESC, property, DEFAULT_NULL_HANDLING);
}
```

所以我们想要根据id进行排序可以这样写:`PageRequest.of(page,size,Sort.by(Sort.Order.asc("id")))`;

八.Kibana的使用

Kibana是一个基于Node.js的ElasticSearch索引库的数据统计工具,可以利用ElasticSearch的聚合功能,生成各种图表. 在开启Kibana之后,可以通过http://localhost:5601进行访问.

1.索引库操作

1. 创建索引库:PUT "索引库名"
2. 查看索引库:GET "索引库名"
3. 删除索引库:DELETE "索引库"

2.类型及映射操作

2.1.创建字段映射

有了索引库,等于就有了数据库中的`database`.接下来就需要建立索引库中的类型了,也就相当于数据库中的表.创建表需要设置字段约束,创建类型也一样,在创建类型时候,需要知道这个类型下有哪些字段,每个字段有哪些约束信息,这个叫做字段映射(Mapping).

```
PUT /索引库名/_mapping/类型名称
{
  "properties": {
    "字段名": {
      "type": "类型",
      "index": true,
      "store": true,
      "analyzer": "分词器"
    }
  }
}
```

类型名称：就是前面将的type的概念，类似于数据库中的表 字段名：任意填写，下面指定许多属性，例如：

- type：类型，可以是text、long、short、date、integer、object等
- index：是否索引，默认为true
- store：是否存储，默认为false
- analyzer：分词器，这里的`ik_max_word`即使用ik分词器

示例

```
PUT fechin/_mapping/goods
{
  "properties": {
    "title": {
      "type": "text",
      "analyzer": "ik_max_word"
    },
    "images": {
      "type": "keyword",
      "index": "false"
    },
    "price": {
      "type": "float"
    }
  }
}
```

上述案列中,就给fechin这个索引库添加了一个名为goods的类型,并且在类型中设置了3个字段.title,images,price

2.2.查看映射关系

```
GET /索引库名/_mapping
```

示例

```
GET /fechin/_mapping
```

2.3.附录:映射属性

2.3.1.type

- String类型:
  - text:可分词,不可参与聚合
  - keyword:不可分词,数据会作为完整字段进行匹配,可以参与聚合.
- Numerical类型
  - 基本数据类型:long、interger、short、byte、double、float、half_float
  - 浮点数的高精度类型:scaled_float
- Date:日期类型
  - elasticsearch可以对日期格式化为字符串存储,但是建议存储为毫秒值,存储为long,节省空间
- Array:数组类型
  - 进行匹配时,任意一个元素满足,都认为满足

    - 排序时,如果升序则用数组中的最小值来排序,如果降序则利用数组中的最大值来排序
- Object:对象

2.3.2.index

index影响字段的索引情况:

- true:字段也会被索引,则可以用来进行搜索过滤.默认值就是true.
- false:字段不会被索引,不能用来索引

index的值默认值就是true,也就是说你不进行任何配置,所有字段都会被索引.

2.3.3.store

ElasticSearch在创建文档索引时候,会将文档中的原始数据备份,保存到一个叫做`_souce`的属性中,而且我们可以通过过滤_source来选择哪些要显示,哪些不显示.

而如果设置store为true,就会在_source以外额外创建一份数据,多余,因此一般我们将store设置为false.事实上,store的默认值就是false.

2.3.4.boost

权重,新增数据时,可以指定数据的权重,权重越高,得分越高,排名越靠前.

2.4.一次创建索引库和类型

```
put /索引库名
{
    "settings":{
        "索引库属性名":"索引库属性值"
    },
    "mappings":{
        "类型名":{
            "properties":{
                "字段名":{
                    "映射属性名":"映射属性值"
                }
            }
        }
    }
}
```

示例

```
PUT /fechin
{
  "settings": {},
  "mappings": {
    "goods": {
      "properties": {
        "title": {
          "type": "text",
          "analyzer": "ik_max_word"
```

```
            }
          }
        }
      }
    }
```

3.文档操作

3.1.新增文档

通过POST请求,可以向一个已经存在的索引库中添加文档数据

```
POST /索引库名/类型名
{
    "key":"value"
}
```

示例:

```
POST /fechin/goods/
{
    "title":"小米手机",
    "images":"http://image.leyou.com/12479122.jpg",
    "price":2699.00
}
```

响应

```
{
  "_index": "fechin",
  "_type": "goods",
  "_id": "r9c1KGMBIhaxtY5rlRKv",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 3,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 0,
  "_primary_term": 2
}
```

可以看到结果显示为"created",创建成功.

在响应结果中,有一个_id字段,这个就是这条文档数据的唯一标识,以后的增删改查都依赖这个id作为唯一标识,

可以看到id的值为`r9c1KGMBIhaxtY5rlRKv`,这里我们新增时,没有指定id,所以是ELasticSearch帮我们随机生成的id.

3.2.查看文档

示例

```
GET /fechin/goods/r9c1KGMBIhaxtY5rlRKv
```

响应

```
{
  "_index": "fechin",
  "_type": "goods",
  "_id": "r9c1KGMBIhaxtY5rlRKv",
  "_version": 1,
  "found": true,
  "_source": {
    "title": "小米手机",
    "images": "http://image.leyou.com/12479122.jpg",
    "price": 2699
  }
}
```

`_source`:源文档信息,所有的数据都在里面

`_id`:这条文档的唯一标识.

3.3.新增文档并自定义id

```
POST /索引库名/类型/id值
{
    ...
}
```

示例

```
POST /fechin/goods/2
{
    "title":"大米手机",
    "images":"http://image.leyou.com/12479122.jpg",
    "price":2899.00
}
```

响应

```
{
  "_index": "fechin",
  "_type": "goods",
  "_id": "2",
  "_score": 1,
  "_source": {
    "title": "大米手机",
    "images": "http://image.leyou.com/12479122.jpg",
    "price": 2899
  }
}
```

3.4.修改文档

把刚才新增的请求方式改为PUT,就是修改,不过修改必须指定id

- id对应文档存在,则修改
- id对应文档不存在,则新增.

```
PUT /heima/goods/3
{
    "title":"超大米手机",
    "images":"http://image.leyou.com/12479122.jpg",
    "price":3299.00,
    "stock": 100,
    "saleable":true
}
```

响应

```
{
  "_index": "heima",
  "_type": "goods",
  "_id": "3",
  "_version": 2,
  "result": "updated",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 2,
  "_primary_term": 1
}
```

可以看到结果是:updated,显然是更新数据.

3.5.删除文档

```
DELETE /索引库名/类型名/id值
```

3.6.查询文档

3.6.1.基本查询

```
GET /索引库名/_search
{
    "query":{
        "查询类型":{
            "查询条件":"查询条件值"
        }
    }
}
```

这里的query代表的是一个查询对象,里面可以有不同的查询属性

- 查询类型:
    - 列如:`match_all`,`match`,`term`,`range`等
- 查询条件:查询条件会根据查询类型的不同,写法也不同.

3.6.2.查询所有

```
GET /fechin/_search
{
    "query":{
        "match_all": {}
    }
}
```

3.6.3.匹配查询

```
GET /fechin/_search
{
    "query":{
        "match":{
            "title":"小米电视"
        }
    }
}
```

注意:match类型查询,会把查询条件进行分词,然后进行查询,多个词条自己是or关系

```
GET /goods/_search
{
    "query":{
        "match":{
            "title":{"query":"小米电视","operator":"and"}
        }
    }
}
```

这样我们就实现了更精确的查找,只有同时包含`小米`和`电视`的词条才会被搜索到.

3.6.4.词条匹配

```
GET /fechin/_search
{
    "query":{
        "term":{
            "price":2699.00
        }
    }
}
```

term查询被用于精确值匹配.这些精确值可能是数字,时间,布尔或者那些未分词的字符串

3.6.5.模糊查询

```
GET /fechin/_search
{
  "query": {
    "fuzzy": {
      "title": "appla"
    }
  }
}
```

fuzzy查询是`term`查询的模糊等价,它允许用户搜索词条与实际词条出现偏差,但是偏差的编辑距离不得超过2;

我们也可以通过fuzziness来指定允许的编辑距离;

```
GET /fechin/_search
{
  "query": {
    "fuzzy": {
        "title": {
            "value":"appla",
            "fuzziness":1
        }
    }
  }
}
```

3.6.6.排序查询

```
GET /fechin/_search
{
  "query": {
    "match": {
      "title": "小米手机"
    }
  },
  "sort": [
    {
      "price": {
        "order": "desc"
      }
    }
  ]
```

3.6.7.分页查询

```
GET /fechin/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "price": {
        "order": "asc"
      }
    }
  ],
  "from": 3,
  "size": 3
}
```

from:开始位置,size:每页大小

3.6.8.高亮

```
GET /fechin/_search
{
  "query": {
    "match": {
      "title": "手机"
    }
  },
  "highlight": {
    "pre_tags": "<em>",
    "post_tags": "</em>",
    "fields": {
      "title": {}
    }
  }
}
```

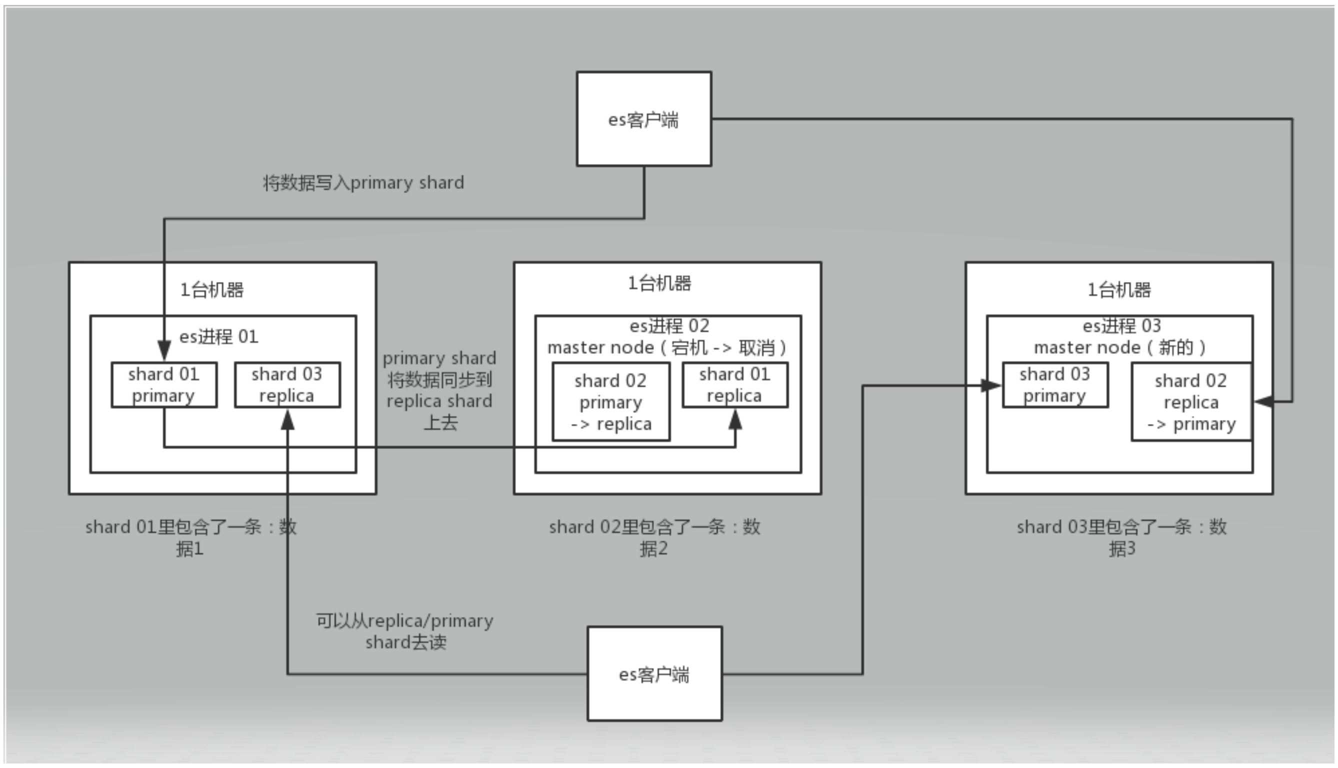
在使用match查询的同时,加上一个highlight属性:

- pre_tags：前置标签
- post_tags：后置标签
- fields：需要高亮的字段

- title：这里声明title字段需要高亮，后面可以为这个字段设置特有配置，也可以空结果.

九.ElasticSearch架构

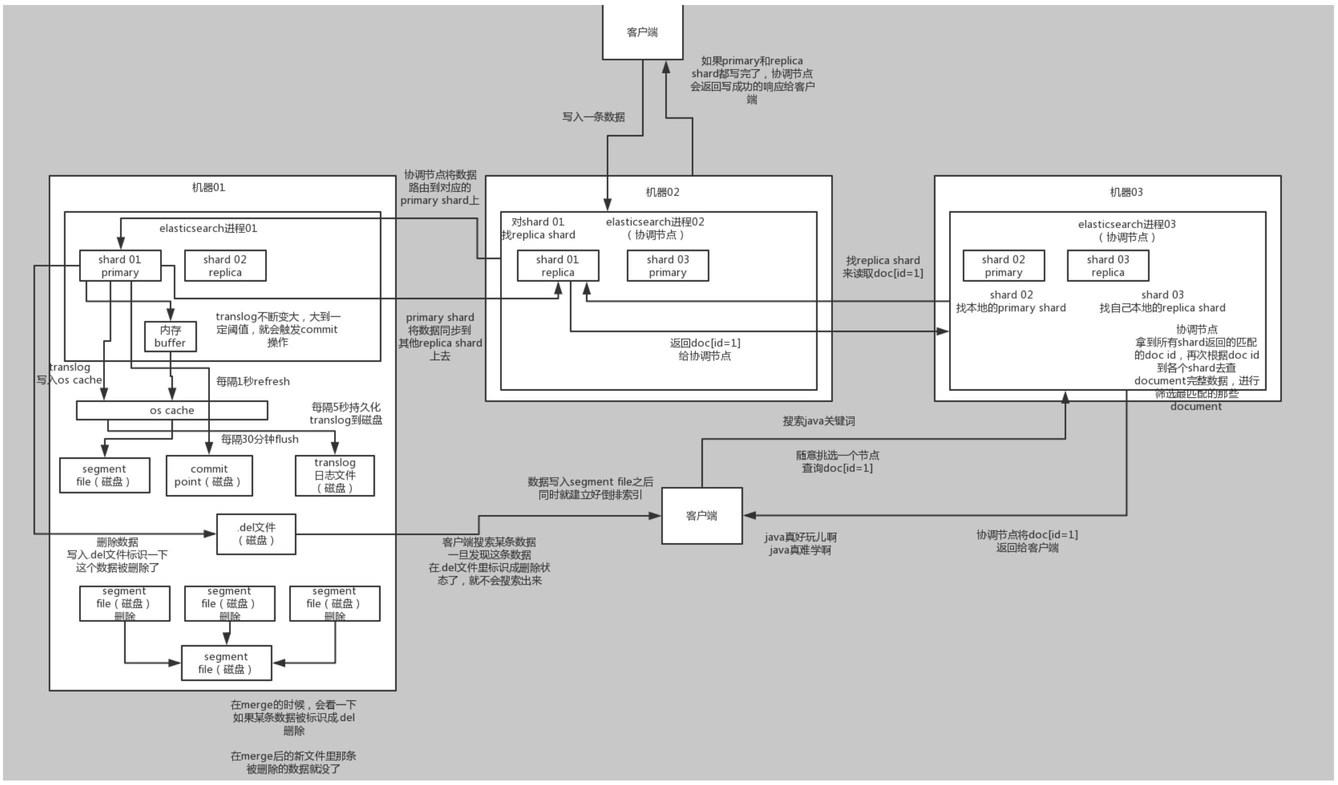
1.ElasticSearch分布式架构原理



es客户端
将数据写入primary shard
1台机器
es进程 01
shard 01 primary
shard 03 replica
primary shard 将数据同步到 replica shard 上去
1台机器
es进程 02 master node（宕机 -> 取消）
shard 02 primary -> replica
shard 01 replica
1台机器
es进程 03 master node（新的）
shard 03 primary
shard 02 replica -> primary
shard 01里包含了一条：数据1
shard 02里包含了一条：数据2
shard 03里包含了一条：数据3
可以从replica/primary shard去读
es客户端

你创建一个索引,这个索引可以拆分成多个shard,每个shard存储部分数据.接着就是这个shard的数据实际上是由多个备份,就是说每个shard都有一个primary shard,负责写入数据,但是还有几个replica shard.primary shard写入数据之后,会将数据同步到其他几个replica shard上去.通过这个replica方案,每个shard的数据都有多个备份,如果某个机器宕机了,没关系,还有别的数据副本在别的机器上.ElasticSearch集群多个节点,会自动选举一个节点为master节点,这个master节点其实就是干一些管理工作,比如维护索引元数据.负责切换primary shard和replica shard身份等.要是master节点宕机了,那么会重新选举一个节点为master节点.如果是非master节点宕机,那么会由master节点让那个宕机节点上的primary shard的身份转移到其他机器上的replica shard.修复完宕机的那个机器后,重启之后,master节点会控制将缺失的replica shard 分配过去.同步后续修改的数据子类的,让集群恢复正常.

2.ElasticSearch写入数据的原理


客户端
写入一条数据
如果primary和replica shard都写完了，协调节点会返回写成功的响应给客户端
机器01
elasticsearch进程01
shard 01 primary
shard 02 replica
内存 buffer
translog不断变大，大到一定阈值，就会触发commit操作
translog 写入os cache
每隔1秒refresh
os cache
每隔5秒持久化translog到磁盘
每隔30分钟flush
segment file（磁盘）
commit point（磁盘）
translog 日志文件（磁盘）
.del文件（磁盘）
删除数据 写入.del文件标识一下 这个数据被删除了
segment file（磁盘） 删除
segment file（磁盘） 删除
segment file（磁盘） 删除
segment file（磁盘）
在merge的时候，会看一下如果某条数据被标识成.del删除
在merge后的新文件里那条被删除的数据就没了
协调节点将数据路由到对应的primary shard上
primary shard将数据同步到其他replica shard上去
机器02
对shard 01 找replica shard
elasticsearch进程02（协调节点）
shard 01 replica
shard 03 primary
返回doc[id=1] 给协调节点
找replica shard 来读取doc[id=1]
机器03
elasticsearch进程03（协调节点）
shard 02 primary
shard 03 replica
shard 02 找本地的primary shard
shard 03 找自己本地的replica shard
协调节点 拿到所有shard返回的匹配的doc id，再次根据doc id到各个shard去查document完整数据，进行筛选最匹配的那些document
数据写入segment file之后同时就建立好倒排索引
客户端搜索某条数据 一旦发现这条数据在.del文件里标识成删除状态了，就不会搜索出来
搜索java关键词
随意挑选一个节点 查询doc[id=1]
客户端
java真好玩儿啊 java真难学啊
协调节点将doc[id=1] 返回给客户端