Elasticsearch 讲师:梁开权(逍遥)

课程目标

- 了解ES中的基本概念
- 掌握RESTFul操作ES的CRUD
- 掌握Spring Data Elasticsearch操作ES

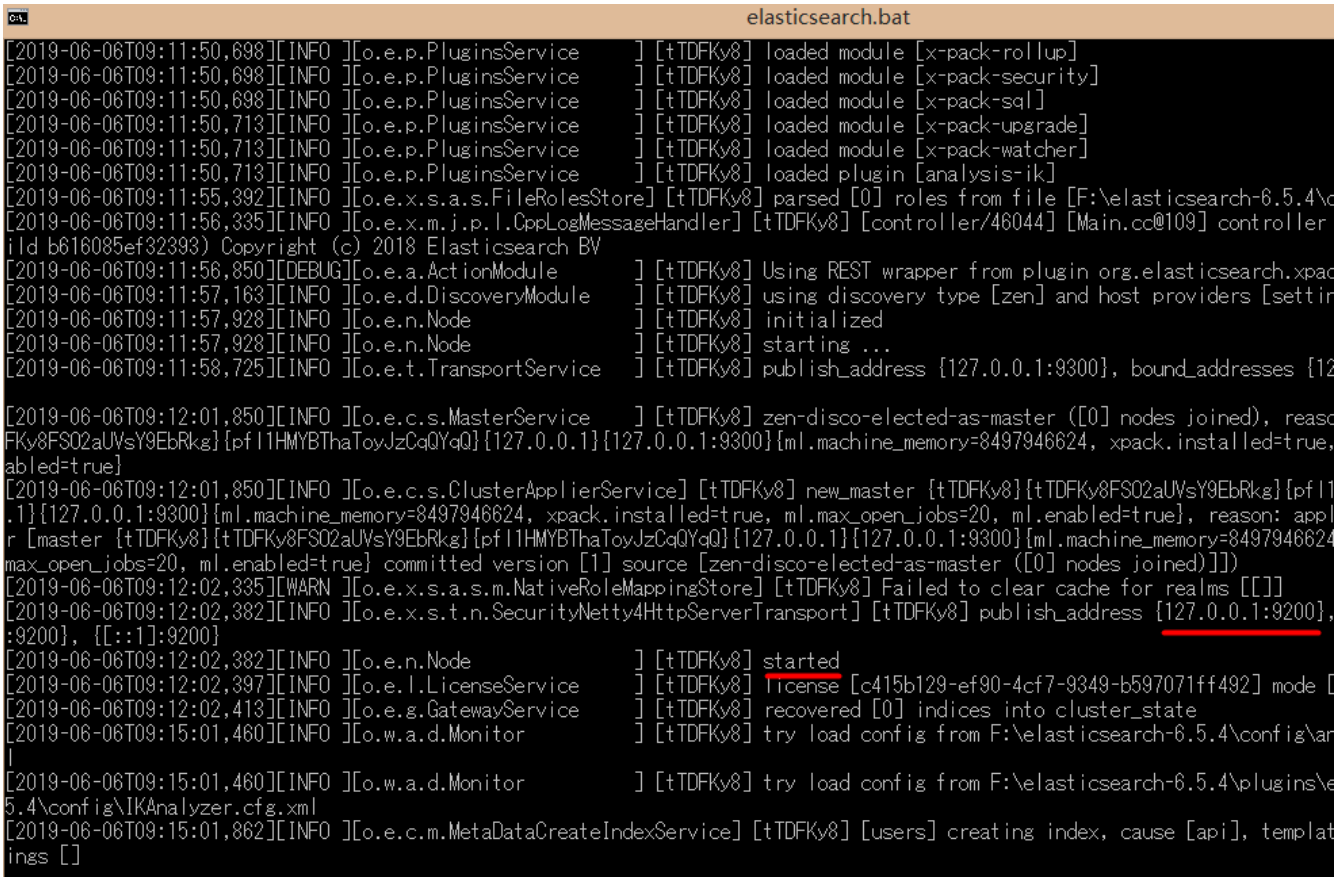
简介

Elasticsearch是一个基于Lucene的搜索服务器。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，能够达到实时搜索，稳定，可靠，快速，安装使用方便。

我们建立一个网站或应用程序，并要添加搜索功能，但是想要完成搜索工作的创建是非常困难的。我们希望搜索解决方案要运行速度快，我们希望能有一个零配置和一个完全免费的搜索模式，我们希望能够简单地使用JSON通过HTTP来索引数据，我们希望我们的搜索服务器始终可用，我们希望能够从一台开始并扩展到数百台，我们要实时搜索，我们要简单的多用户，我们希望建立一个云的解决方案。因此我们利用Elasticsearch来解决所有这些问题及可能出现的更多其它问题。

安装和运行

该软件是基于Java编写的解压即用的软件，只需要有Java的运行环境即可，把压缩包解压后，进入到bin目录运行elasticsearch.bat，出现以下界面，表示成功启动服务器



浏览器输入：http://localhost:9200，看到浏览器输出服务器的信息，表示安装成功，可以使用了

后台启动

使用**安装目录/bin/elasticsearch-service.bat**程序可以把Elasticsearch安装后服务列表中，以后我们可以在服务列表来启动该程序，也可以设置成开机启动模式

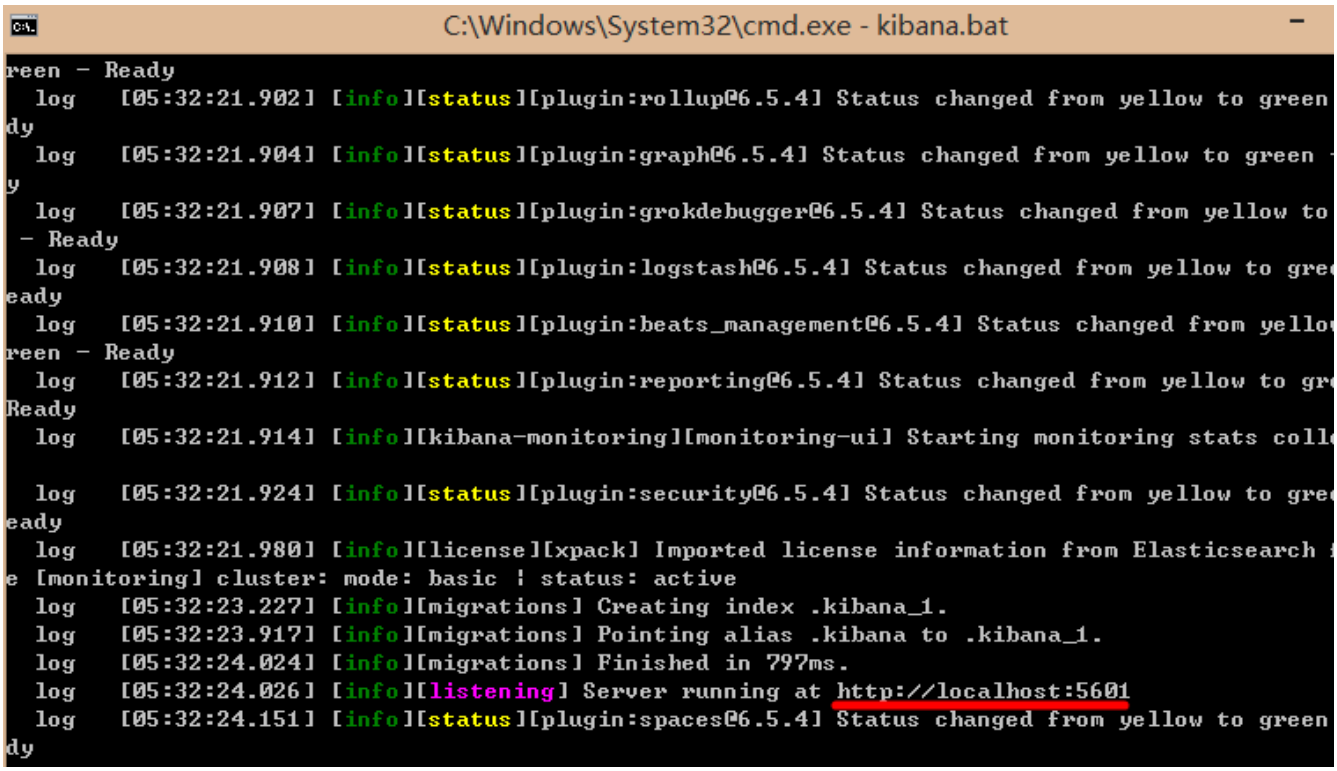
```
F:\elasticsearch-6.5.4\bin>elasticsearch-service.bat install
Installing service      :  "elasticsearch-service-x64"
Using JAVA_HOME (64-bit):  "D:\Java\jdk1.8.0_152"
-Xms1g;-Xmx1g;-XX:+UseConcMarkSweepGC;-XX:CMSInitiatingOccupan
upancyOnly;-XX:+AlwaysPreTouch;-Xss1m;-Djava.awt.headless=true
e;-XX:-OmitStackTraceInFastThrow;-Dio.netty.noUnsafe=true;-Dio
netty.recycler.maxCapacityPerThread=0;-Dlog4j.shutdownHookEnab
java.io.tmpdir=C:\Users\Bunny\AppData\Local\Temp\elasticsearch
eapDumpPath=data;-XX:ErrorFile=logs/hs_err_pid%p.log;-XX:+Prin
+PrintTenuringDistribution;-XX:+PrintGCApplicationStoppedTime;
otation;-XX:NumberOfGCLogFiles=32;-XX:GCLogFileSize=64m
The service 'elasticsearch-service-x64' has been installed.
```

安装head插件（学生可以不安装）

Elasticsearch默认的客户端工具是命令行形式的，操作起来不方便，也不直观看到数据的展示，所以我们需要去安装一个可视化插件，但是这些插件都是基于H5开发的，在谷歌的应用商店中找到**elasticsearch-head**插件，然后安装，使用该插件能比较直观的展示服务器中的数据

安装kibana

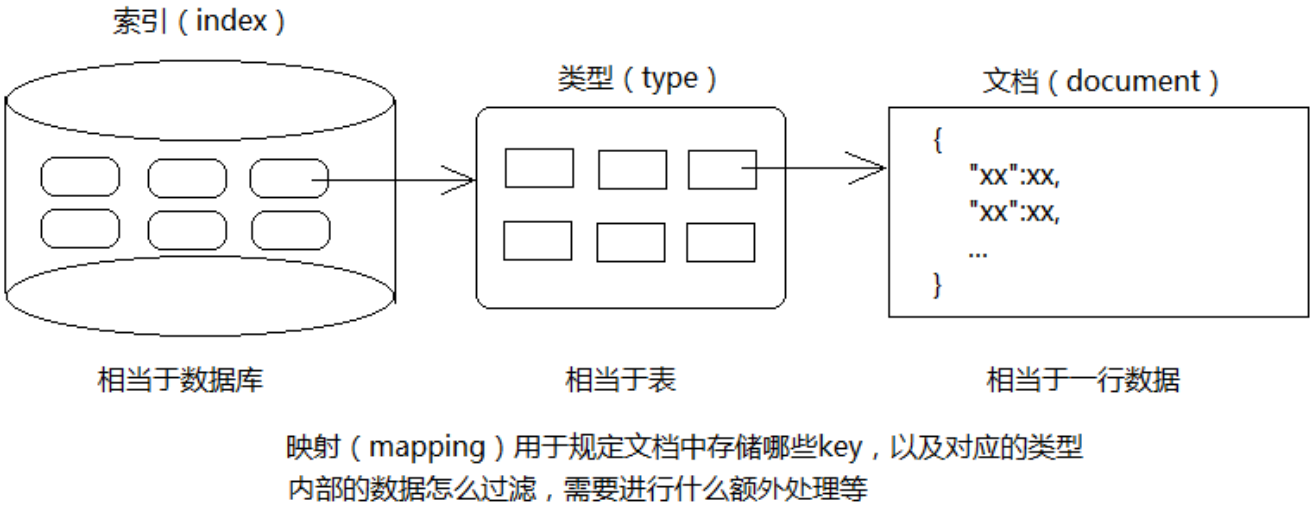
该软件也是解压即用的工具，用于管理和监控Elasticsearch的运作，同时内部包含了客户端工具，支持RESTFul操作Elasticsearch。解压后运行bin/kibana.bat，看到启动成功的端口号即可以使用浏览器来使用了



浏览器输入：http://localhost:5601

概念名词

数据存储图



注意：从Elasticsearch6开始一个索引里面只能有一个类型

主要组件

- 索引

  ES将数据存储于一个或多个索引中，索引是具有类似特性的文档的集合。类比传统的关系型数据库领域来说，索引相当于SQL中的一个数据库，或者一个数据存储方案(schema)。索引由其名称(必须为全小写字符)进行标识，并通过引用此名称完成文档的创建、搜索、更新及删除操作。一个ES集群中可以按需创建任意数目的索引。

- 类型

  类型是索引内部的逻辑分区(category/partition)，然而其意义完全取决于用户需求。因此，一个索引内部可定义一个或多个类型(type)。一般来说，类型就是为那些拥有相同的域的文档做的预定义。例如，在索引中，可以定义一个用于存储用户数据的类型，一个存储日志数据的类型，以及一个存储评论数据的类型。类比传统的关系型数据库领域来说，类型相当于表。

- 映射

  Mapping,就是对索引库中索引的字段名称及其数据类型进行定义，类似于mysql中的表结构信息。不过es的mapping比数据库灵活很多，它可以动态识别字段。一般不需要指定mapping都可以，因为es会自动根据数据格式识别它的类型，如果你需要对某些字段添加特殊属性（如：定义使用其它分词器、是否分词、是否存储等），就必须手动添加mapping。

  我们在es中添加索引数据时不需要指定数据类型，es中有自动影射机制，字符串映射为string，数字映射为long。通过mappings可以指定数据类型是否存储等属性

- 文档

  文档是Lucene索引和搜索的原子单位，它是包含了一个或多个域的容器，基于JSON格式进行表示。文档由一个或多个域组成，每个域拥有一个名字及一个或多个值，有多个值的域通常称为多值域。每个文档可以存储不同的域集，但同一类型下的文档至应该有某种程度上的相似之处。

分片和副本

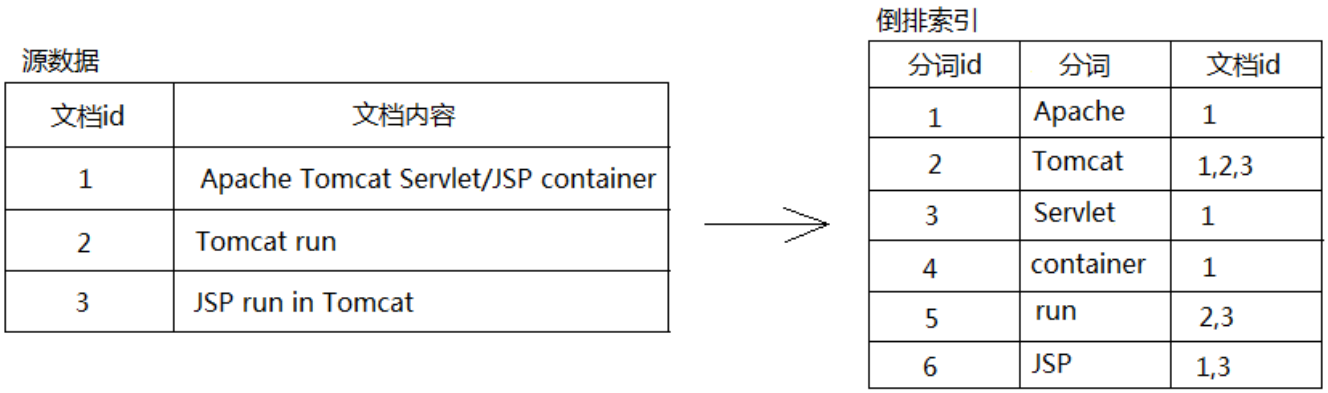
ES的分片(shard)机制可将一个索引内部的数据分布地存储于多个节点，它通过将一个索引切分为多个底层物理的Lucene索引完成索引数据的分割存储功能，这每一个物理的Lucene索引称为一个分片(shard)。每个分片其内部都是一个全功能且独立的索引，因此可由集群中的任何主机存储。创建索引时，用户可指定其分片的数量，默认数量为5个。

Shard有两种类型：primary和replica，即主shard及副本shard。Primary shard用于文档存储，每个新的索引会自动创建5个Primary shard，当然此数量可在索引创建之前通过配置自行定义，不过，一旦创建完成，其Primary shard的数量将不可更改。Replica shard是Primary Shard的副本，用于冗余数据及提高搜索性能。每个Primary shard默认配置了一个Replica shard，但也可以配置多个，且其数量可动态更改。ES会根据需要自动增加或减少这些Replica shard的数量。

分词器

把文本内容按照标准进行切分，默认的是standrad，该分词器按照单词切分，内容转变为小写，去掉标点，遇到每个中文字符都当成1个单词处理，后面会安装中文分词器插件（ik）

倒排索引



基本操作

索引操作（了解）

建表索引，相当于在是在建立数据库

建立索引

```
语法：PUT /索引名
在没有特殊设置的情况下，默认有5个分片，1个备份，也可以通过请求参数的方式来指定
参数格式：
{
    "settings": {
        "number_of_shards": 5, //设置5个片区
        "number_of_replicas": 1 //设置1个备份
    }
}
```

删除索引

```
语法：DELETE /索引名
```

文档操作

添加文档

```
语法：PUT /索引名/类型名/文档ID
{
    field1: value1,
    field2: value2,
    ...
}

注意：当索引/类型/映射不存在时，会使用默认设置自动添加
```

每一个文档都内置以下字段

> _index：所属索引
>
> _type：所属类型
>
> _id：文档ID，若不指定则需要使用POST请求，会自动生成UUID值
>
> _version：乐观锁版本号
>
> _source：数据内容

查询文档

```
语法：
根据ID查询 -> GET /索引名/类型名/文档ID
```

查询的结果中包含以下字段

> took：耗时
>
> _shards.total：分片总数
>
> hits.total：查询到的数量
>
> hits.max_score：最大匹配度
>
> hits.hits：查询到的结果
>
> hits.hits._score：匹配度/匹配值

更新文档

```
语法：
POST /索引名/类型名/文档ID
{
    "doc": {
        field1: value1,
        field2: value2,
        ...
    }
}

该操作实质上是使用新的文档替换就的文档
```

删除文档

```
语法：DELETE /索引名/类型名/文档ID
注意：这里的删除并且不是真正意义上的删除，仅仅是清空文档内容而已
```

结果排序

```
参数格式：
{
    "sort": [
        {field: 排序规则},
        ...
    ]
}
```

分页查询

```
参数格式：
{
    "from": start,
    "size": pageSize
}
```

```
阶段练习：
1：使用POST方法往rbac/employee中加入一个文档，文档具有id,name,age,deptId属性
2：使用PUT方法基于文档ID更新一个文档
3：使用GET方法基于文档ID查询一个文档
4：使用DELETE方法基于文档ID删除一个文档
5：查询rbac/employee中的所有文档，按照年龄排序
6：分页查询rbac/employee中的文档，每页显示3个，显示第2页
```

高级查询

Elasticsearch基于JSON提供完整的查询DSL（Domain Specific Language：领域特定语言）来定义查询。

```
基本语法：
GET /索引名/类型名/_search
```

一般都是需要配合查询参数来使用的，配合不同的参数有不同的查询效果

参数配置项可以参考博客：https://www.jianshu.com/p/6333940621ec

查询所有

参数格式：

```
{
    "query": {"match_all": {}}
}
```

含义：
query：配置查询类型，系统会对查询结果评分
match_all：匹配所有

投影查询

参数格式：

```
{
    "_source": [field1, field2, ...]
}
```

阶段练习:
1：查询rbac/employee中所有文档
2：查询rbac/employee中所有文档，只显示name,age字段

检索查询

参数格式：

```
{
    "query": {
        检索方式: {field: value}
    }
}
```

检索方式：
term表示精确匹配，value值不会被分词器拆分，按照倒排索引匹配，一般用于filter中
match_phrase表示短语检索，value值不会被分词器拆分，直接去倒排索引中匹配
match表示全文检索，value值会被分词器拆分，然后去倒排索引中匹配

逻辑查询

参数格式：

```
{
    "query": {
        "bool": {
            逻辑规则: [
                {检索方式: {field: value}},
                ...
            ]
        }
    }
}
```

逻辑规则：must / should / must_not，相当于and / or / not

范围查询

```
参数格式：
{
    "query": {
        "bool": {
            "filter": {
                检索方式: {
                        field: value
                },
                "range": {
                    field: {比较规则: value, ...}
                }
            }
        }
    }
}

range：范围过滤
比较规则：gt / gte / lt / lte 等
注意：filter表示在查询的结果中做过滤，该操作不涉及评分，有缓存，适用于完全精确匹配，范围查询
```

关于filter的更多认知推荐大家读这篇博客：https://blog.csdn.net/laoyang360/article/details/80468757

```
阶段练习：
1：查询rbac/employee中所有name含有zhang的文档
2：查询rbac/employee中所有name含有zhang并且age=26的文档
3：查询rbac/employee中所有name含有zhang或者age=33的文档
4：查询rbac/employee中所有name含有zhang并且26<=age<=33的文档
```

分组查询

```
参数格式：
{
    "size": 0,
    "aggs": {
        自定义分组字段: {
            "terms": {"field": 分组字段, "order": {自定义统计字段:排序规则}},
            "aggs": { //分组后的统计查询，相当于MySQL分组函数查询
                自定义统计字段: {
                    分组运算: {
                        "field": 统计字段
                    }
                }
            },
            "size": 10 //默认显示10组
        }
    }
}
```

分组运算：avg / sum / min / max / value_count / stats(执行以上所有功能的)
注意：这里是size=0其目的是为了不要显示hit内容，专注点放在观察分组上

阶段练习：
1：按照部门编号分组，统计文档数量
2：按照部门编号分组，统计各部门的平均年龄
3：按照部门编号分组，统计各部门的年龄状态

批处理

当需要集中的批量处理文档时，如果依然使用传统的操作单个API的方式，将会浪费大量网络资源，Elasticsearch为了提高操作的性能，专门提供了批处理的API

mget批量查询

```
语法：
GET /索引名/类型/_mget
{
    "docs": [
        {"_id": 文档ID},
        ...
    ]
}
```

bulk批量增删改

```
语法：
POST /索引名/类型/_bulk
{动作:{"_id": 文档ID}
{...}
{动作:{"_id": 文档ID}
{...}

动作：create / update / delete，其中delete只有1行JSON，其他操作都是有2行JSON，并且JSON不能格式化
```

IK分词器插件

默认的分词器是按照单词边界来拆分词语的，对于英文来讲是完全够用的，但是对于中文来说的话，并不是一个字一个词，有的是多个字一个词，所以默认的分词器就显得不满足分词的需求，这里给大家推荐支持中文的分词器IK

安装

直接把压缩文件中的内容解压，然后放在elasticsearch/plugins下，然后重启即可

感受IK分词器

```
GET /rbac/_analyze
{
  "text":"为了完成删除匹配到的所有数据库，然后跑路，去学习怎么写代码",
  "analyzer":"ik_max_word"
}

IK插件中有2个分词器
ik_max_word：细粒度分词
ik_smart：粗粒度分词
```

使用IK分词器

文档中的字段的数据类型和是否分词等信息都是在mapping（映射）中配置的，通过配置mapping来修改文档中字段默认的分词器

```
查看类型的映射语法：
GET /索引名/类型名/_mapping

如：GET /rbac/employee/_mapping
```

在映射中我们可以看到Elasticsearch是支持数据类型的，而且能通过给定的值来自动识别出数据的类型，这个是归功于Elasticsearch的动态映射特征，其中我们可以看到，name字段的类型的是text，也只有type=text的字段才能分词

那么这里我们没有看到映射中包含使用IK分词器，那么我则必须通过手动创建映射的方式来配置

步骤：
1：先删除旧的索引，否则会有冲突
DELETE /rbac
2：重新创建索引库，并且通过参数指定映射
PUT /rbac
```
{
  "mappings": {
    "employee": {
      "properties": {
        "id": {"type": "long"},
        "name": {
            "type": "text",
            "analyzer": "ik_max_word",
            "search_analyzer": "ik_max_word",
            "fields": {
                "keyword": {
                    "type": "keyword",
                    "ignore_above": 256
                }
            }
        },
        "age": {"type":"integer"},
        "deptId": {"type":"long"}
      }
```

```
    }
  }
}
3：再次查看employee类型的映射信息

type的类型有以下几种：
double / long / integer / text / keyword / date / binary

text和keyword区别：
1.都是表示字符串类型
2.text会分词，然后再存储到倒排索引库中
3.keyword不分词，然后直接存储到倒排索引库中
```

Spring Data Elasticsearch

准备环境

导入依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
</parent>

<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <!--SpringBoot整合Spring Data Elasticsearch的依赖-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

编写domain

```
/**
  @Document：配置操作哪个索引下的哪个类型
  @Id：标记文档ID字段
  @Field：配置映射信息，如：分词器
 */
@Getter@Setter@ToString
@NoArgsConstructor
@AllArgsConstructor
@Document(indexName="rbac", type="employee")
public class Employee implements Serializable {
    @Id
    private Long id;

    @Field(analyzer="ik_max_word", searchAnalyzer="ik_max_word", type=FieldType.Text)
    private String name;
    private Integer age;
    private Long deptId;
}
```

配置连接信息

```
#application.properties
# 配置集群名称，名称写错会连不上服务器，默认elasticsearch
spring.data.elasticsearch.cluster-name=elasticsearch
# 配置集群节点
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

组件介绍

```
ElasticsearchTemplate：框架封装的用于便捷操作Elasticsearch的模板类
ElasticsearchRepository：框架封装的用于便捷完成CRUD的接口

NativeSearchQueryBuilder：用于生成查询条件的构建器，需要去封装各种查询条件
QueryBuilder：该接口表示一个查询条件，其对象可以通过QueryBuilders工具类中的方法快速生成各种条件
  boolQuery()：生成bool条件，相当于 "bool": { }
  matchQuery()：生成match条件，相当于 "match": { }
  rangeQuery()：生成range条件，相当于 "range": { }
AbstractAggregationBuilder：用于生成分组查询的构建器，其对象通过AggregationBuilders工具类生成
Pageable：表示分页参数，对象通过PageRequest.of(页数, 容量)获取
SortBuilder：排序构建器，对象通过SortBuilders.fieldSort(字段).order(规则)获取
```

ElasticsearchTemplate

该模板类，封装了便捷操作Elasticsearch的模板方法，包括 索引 / 映射 / CRUD 等底层操作和高级操作，该对象用起来会略微复杂些，尤其是对于查询，还需要把查询到的结果自己封装对象

```
//该对象已经由SpringBoot完成自动配置，直接注入即可
@Autowired
private ElasticsearchTemplate elasticsearchTemplate;
```

ElasticsearchRepository

该接口是框架封装的用于操作Elastsearch的高级接口，只要我们自己的写个接口去继承该接口就能直接对Elasticsearch进行CRUD操作

```
/**
  泛型1：domain的类型
  泛型2：@Id的类型
  该接口直接该给Spring，底层会使用JDK代理的方式创建对象，交给容器管理
 */
@Repository
public interface EmployeeSearch extends ElasticsearchRepository<Employee, Long> {
    // 还能使用JPA规范命名方法，来做高级查询，了解即可
    // List<Employee> findByName(String name);
}
```

一般情况下，ElasticsearchTemplate和ElasticsearchRepository是分工合作的，ElasticsearchRepository用于完成CRUD操作，如果遇到底层操作和聚合查询则要使用ElasticsearchTemplate

实例代码

```
// 新增或者更新一个文档
@Test
public void testSaveOrUpdate() throws Exception {
    Employee emp = new Employee(null, "逍遥", 10, 5L);
    // 没有ID则新增，有ID则更新
    // emp.setId(...);
    employeeSearch.save(emp);
}

// 删除一个文档
@Test
public void testDelete() throws Exception {
    Employee emp = new Employee();
    emp.setId(11L);
    employeeSearch.delete(emp);
}

// 根据ID查询一个文档
@Test
public void testGet() throws Exception {
    Employee emp = employeeSearch.findById(1L).get();;
    System.err.println(emp);
}
```

```
// 查询所有文档
@Test
public void testList() throws Exception {
    Iterable<Employee> list = employeeSearch.findAll();
    list.forEach(System.err::println);
}

// 分页查询文档，显示第2页，每页显示3个，按照id升序排列
@Test
public void testPage() throws Exception {
    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
    builder.withPageable(PageRequest.of(1, 3))
        .withSort(SortBuilders.fieldSort("id").order(SortOrder.ASC));

    Page<Employee> page = employeeSearch.search(builder.build());
    System.out.println(page.getTotalElements());
    System.out.println(page.getTotalPages());
    page.forEach(System.err::println);
}

// 查询所有name含有zhang的文档
@Test
public void testQuery1() throws Exception {
    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
    builder.withQuery(QueryBuilders.matchQuery("name", "zhang"));

    Page<Employee> page = employeeSearch.search(builder.build());
    System.out.println(page.getTotalElements());
    System.out.println(page.getTotalPages());
    page.forEach(System.err::println);
}

// 查询所有name含有zhang或者age=33的文档
@Test
public void testQuery3() throws Exception {
    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
    builder.withQuery(
        QueryBuilders.boolQuery()
        .should(QueryBuilders.matchQuery("name", "zhang"))
        .should(QueryBuilders.matchQuery("age", "33"))
    );

    Page<Employee> page = employeeSearch.search(builder.build());
    System.out.println(page.getTotalElements());
    System.out.println(page.getTotalPages());
    page.forEach(System.err::println);
}

// 查询所有name含有wang并且30<=age<=32的文档
@Test
public void testQuery4() throws Exception {
    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
```

```
    builder.withQuery(
        QueryBuilders.boolQuery().must(QueryBuilders.matchQuery("name", "wang"))
        .filter(QueryBuilders.rangeQuery("age").gte(30).lte(32))
    );

    Page<Employee> page = employeeSearch.search(builder.build());
    System.out.println(page.getTotalElements());
    System.out.println(page.getTotalPages());
    page.forEach(System.err::println);
}

// 按照部门编号分组，统计文档数量
@Test
public void testQuery5() throws Exception {
    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
    builder.addAggregation(AggregationBuilders.terms("groupDept").field("deptId"));
    // res -> res.getAggregations()是lambda表达式
    Aggregations aggs = elasticsearchTemplate.query(builder.build(), res ->
res.getAggregations());
    LongTerms terms = (LongTerms) aggs.asMap().get("groupDept");
    for (LongTerms.Bucket bucket : terms.getBuckets()) {
        System.out.println(bucket.getDocCount());
    }
}

// 按照部门编号分组，统计各部门的平均年龄
@Test
public void testQuery6() throws Exception {
    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
    builder.addAggregation(
        AggregationBuilders.terms("groupDept").field("deptId")
        .subAggregation(AggregationBuilders.avg("ag_age").field("age"))
    );

    Aggregations aggs = elasticsearchTemplate.query(builder.build(), res ->
res.getAggregations());
    LongTerms terms = (LongTerms) aggs.asMap().get("groupDept");
    for (LongTerms.Bucket bucket : terms.getBuckets()) {
        InternalAvg agAge = (InternalAvg) bucket.getAggregations().asMap().get("ag_age");
        System.out.println(agAge.getValue());
    }
}

// 按照部门编号分组，统计各部门的年龄状态
@Test
public void testQuery7() throws Exception {
    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
    builder.addAggregation(
        AggregationBuilders.terms("groupDept").field("deptId")
        .subAggregation(AggregationBuilders.stats("stat_age").field("age"))
    );

    Aggregations aggs = elasticsearchTemplate.query(builder.build(), res ->
```

```
res.getAggregations());
    LongTerms terms = (LongTerms) aggs.asMap().get("groupDept");
    for (LongTerms.Bucket bucket : terms.getBuckets()) {
        InternalStats stats = (InternalStats)
bucket.getAggregations().asMap().get("stat_age");
        System.out.println(stats.getCount());
        System.out.println(stats.getAvg());
        System.out.println(stats.getMax());
        System.out.println(stats.getMin());
        System.out.println(stats.getSum());
        System.out.println("==================");
    }
}
```