16、Spring Boot与检索

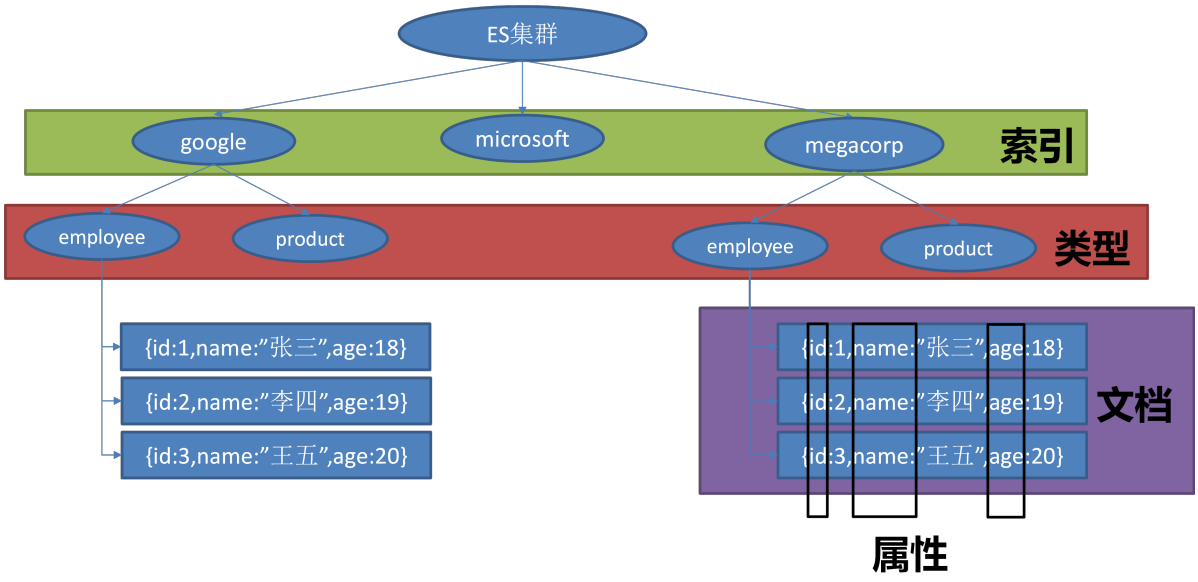
2020年6月25日 8:44

- 1、检索

  我们的应用经常需要添加检索功能，开源的 ElasticSearch 是目前全文搜索引擎的首选。他可以快速的存储、搜索和分析海量数据。Spring Boot通过整合Spring Data ElasticSearch为我们提供了非常便捷的检索功能支持；

  Elasticsearch是一个分布式搜索服务，提供Restful API，底层基于Lucene，采用多shard（分片）的方式保证数据安全，并且提供自动resharding的功能，github等大型的站点也是采用了ElasticSearch作为其搜索服务。

- 2、概念
  - 以 *员工文档* 的形式存储为例：一个文档代表一个员工数据。存储数据到 ElasticSearch 的行为叫做 *索引* ，但在索引一个文档之前，需要确定将文档存储在哪里。
  - 一个 ElasticSearch 集群可以 包含多个 *索引* ，相应的每个索引可以包含多个 *类型* 。 这些不同的类型存储着多个 *文档* ，每个文档又有 多个 *属性* 。
  - 类似关系：
    - 索引-数据库
    - 类型-表
    - 文档-表中的记录
    - 属性-列



- 3、docker安装elasticsearch
  a. 拉取镜像
  docker pull elasticsearch:6.4.3
  b. 启动镜像
  docker run -d --name ES01 -p 9200:9200 -p 9300:9300 -e

"discovery.type=single-node" elasticsearch:6.4.3

c. 进入容器
docker exec -it ES01 /bin/bash

d. 进入配置文件夹
cd config/

e. 修改配置文件
vi elasticsearch.yml

f. 添加配置
http.cors.enabled: true
http.cors.allow-origin: "*"

- 4、测试elasticsearch
  - Elasticsearch 使用 JavaScript Object Notation（或者 *JSON*）作为文档的序列化格式。JSON 序列化为大多数编程语言所支持，并且已经成为 NoSQL 领域的标准格式。 它简单、简洁、易于阅读。
    - 下面这个 JSON 文档代表了一个 user 对象：

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

  - 虽然原始的 user 对象很复杂，但这个对象的结构和含义在 JSON 版本中都得到了体现和保留。在 Elasticsearch 中将对象转化为 JSON 后构建索引要比在一个扁平的表结构中要简单的多。
    对于员工目录，我们将做如下操作：
    每个员工索引一个文档，文档包含该员工的所有信息。
    每个文档都将是 employee *类型* 。
    该类型位于 *索引* megacorp 内。
    该索引保存在我们的 Elasticsearch 集群中。
    - 实践中这非常简单（尽管看起来有很多步骤），我们可以通过一条命令完成所有这些动作：

```
PUT /megacorp/employee/1
{
    "first_name" : "John",
    "last_name" :  "Smith",
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
```

```
}
```

- 注意，路径 /megacorp/employee/1 包含了三部分的信息：
  megacorp
  索引名称
  employee
  类型名称
  1
  特定雇员的ID

- 检索到单个雇员的数据
  - 简单地执行 一个 HTTP GET 请求并指定文档的地址——索引库、类型和 ID。 使用这三个信息可以返回原始的 JSON 文档：
    GET /megacorp/employee/1
  - 返回结果包含了文档的一些元数据，以及 _source 属性，内容是 John Smith 雇员的原始 JSON 文档：

```
{
  "_index" :   "megacorp",
  "_type" :    "employee",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
     "first_name" :  "John",
     "last_name" :   "Smith",
     "age" :         25,
     "about" :       "I love to go rock climbing",
     "interests":  [ "sports", "music" ]
   }
}
```

- 将 HTTP 命令由 PUT 改为 GET 可以用来检索文档，同样的，可以使用 DELETE 命令来删除文档，以及使用 HEAD 指令来检查文档是否存在。如果想更新已存在的文档，只需再次 PUT 。
- 一个简单的搜索，第一个尝试的几乎是最简单的搜索了。我们使用下列请求来搜索所有雇员：
  GET /megacorp/employee/_search
  - 可以看到，我们仍然使用索引库 megacorp 以及类型 employee，但与指定一个文档 ID 不同，这次使用 _search 。返回结果包括了所有三个文档，放在数组 hits 中。一个搜索默认返回十条结果。

```
{
  "took":      6,
  "timed_out": false,
```

```
"_shards": { ... },
"hits": {
  "total":      3,
  "max_score":  1,
  "hits": [
    {
      "_index":         "megacorp",
      "_type":          "employee",
      "_id":            "3",
      "_score":         1,
      "_source": {
        "first_name":  "Douglas",
        "last_name":   "Fir",
        "age":         35,
        "about":       "I like to build cabinets",
        "interests": [ "forestry" ]
      }
    },
    {
      "_index":         "megacorp",
      "_type":          "employee",
      "_id":            "1",
      "_score":         1,
      "_source": {
        "first_name":  "John",
        "last_name":   "Smith",
        "age":         25,
        "about":       "I love to go rock climbing",
        "interests": [ "sports", "music" ]
      }
    },
    {
      "_index":         "megacorp",
      "_type":          "employee",
      "_id":            "2",
      "_score":         1,
      "_source": {
        "first_name":  "Jane",
        "last_name":   "Smith",
        "age":         32,
        "about":       "I like to collect rock albums",
        "interests": [ "music" ]
```

```
        }
      }
    ]
  }
}
```

- 注意：返回结果不仅告知匹配了哪些文档，还包含了整个文档本身：显示搜索结果给最终用户所需的全部信息。

- 接下来，尝试下搜索姓氏为 ``Smith`` 的雇员。为此，我们将使用一个 *高亮* 搜索，很容易通过命令行完成。这个方法一般涉及到一个 *查询字符串* （*query-string*） 搜索，因为我们通过一个URL参数来传递查询信息给搜索接口：

  ```
  GET /megacorp/employee/_search?q=last_name:Smith
  ```

  - 我们仍然在请求路径中使用 _search 端点，并将查询本身赋值给参数 q= 。返回结果给出了所有的 Smith：

```
{
  ...
  "hits": {
    "total":      2,
    "max_score":  0.30685282,
    "hits": [
      {
        ...
        "_source": {
          "first_name":  "John",
          "last_name":   "Smith",
          "age":         25,
          "about":       "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        }
      },
      {
        ...
        "_source": {
          "first_name":  "Jane",
          "last_name":   "Smith",
          "age":         32,
          "about":       "I like to collect rock albums",
          "interests": [ "music" ]
        }
      }
    ]
  }
}
```

- Query-string 搜索通过命令非常方便地进行临时性的即席搜索，但它有自身的局限性（参见 *轻量搜索*）。Elasticsearch 提供一个丰富灵活的查询语言叫做 *查询表达式*， 它支持构建更加复杂和健壮的查询。
  - *领域特定语言*（DSL）， 使用 JSON 构造了一个请求。我们可以像这样重写之前的查询所有名为 Smith 的搜索：

    ```
    GET /megacorp/employee/_search
    {
       "query" : {
          "match" : {
             "last_name" : "Smith"
          }
       }
    }
    ```

  - 返回结果与之前的查询一样，但还是可以看到有一些变化。其中之一是，不再使用 *query-string* 参数，而是一个请求体替代。这个请求使用 JSON 构造，并使用了一个 match 查询（属于查询类型之一，后面将继续介绍）。
- 现在尝试下更复杂的搜索。 同样搜索姓氏为 Smith 的员工，但这次我们只需要年龄大于 30 的。查询需要稍作调整，使用过滤器 *filter*，它支持高效地执行一个结构化查询。

  ```
  GET /megacorp/employee/_search
  {
     "query" : {
        "bool": {
           "must": {
              "match" : {
                 "last_name" : "smith"
              }
           },
           "filter": {
              "range" : {
                 "age" : { "gt" : 30 }
              }
           }
        }
     }
  }
  ```

  - 这部分是一个 range *过滤器*， 它能找到年龄大于 30 的文档，其中 gt 表示_大于_(*great than*)。
- 目前无需太多担心语法问题，后续会更详细地介绍。只需明确我们添加了一个 *过滤器* 用于执行一个范围查询，并复用之前的 match 查询。现在结果只返回了一名员工，叫 Jane Smith，32 岁。

```
{
  ...
  "hits": {
    "total":      1,
    "max_score":  0.30685282,
    "hits": [
      {
        ...
        "_source": {
          "first_name":  "Jane",
          "last_name":   "Smith",
          "age":         32,
          "about":       "I like to collect rock albums",
          "interests": [ "music" ]
        }
      }
    ]
  }
}
```

- 全文搜索一项 传统数据库确实很难搞定的任务。
  搜索下所有喜欢攀岩（rock climbing）的员工：

```
GET /megacorp/employee/_search
{
   "query" : {
      "match" : {
         "about" : "rock climbing"
      }
   }
}
```

  - 显然我们依旧使用之前的 match 查询在`about` 属性上搜索 “rock climbing” 。得到两个匹配的文档：

```
{
  ...
  "hits": {
    "total":      2,
    "max_score":  0.16273327,
    "hits": [
      {
        ...
        "_score":         0.16273327,
        "_source": {
          "first_name":  "John",
```

```
            "last_name":   "Smith",
            "age":         25,
            "about":       "I love to go rock climbing",
            "interests": [ "sports", "music" ]
          }
        },
        {
          ...
          "_score":         0.016878016,
          "_source": {
            "first_name":  "Jane",
            "last_name":   "Smith",
            "age":         32,
            "about":       "I like to collect rock albums",
            "interests": [ "music" ]
          }
        }
      ]
    }
}
```

- 相关性得分

  Elasticsearch 默认按照相关性得分排序，即每个文档跟查询的匹配程度。

  - 第一个最高得分的结果很明显：John Smith 的 about 属性清楚地写着“rock climbing”。
  - 但为什么 Jane Smith 也作为结果返回了呢？原因是她的 about 属性里提到了“rock”。因为只有“rock”而没有“climbing”，所以她的相关性得分低于 John 的。
  - 这是一个很好的案例，阐明了 Elasticsearch 如何 *在* 全文属性上搜索并返回相关性最强的结果。Elasticsearch中的 *相关性* 概念非常重要，也是完全区别于传统关系型数据库的一个概念，数据库中的一条记录要么匹配要么不匹配。
  - 找出一个属性中的独立单词是没有问题的，但有时候想要精确匹配一系列单词或者_短语_。 比如， 我们想执行这样一个查询，仅匹配同时包含“rock” *和* “climbing”，*并且* 二者以短语“rock climbing”的形式紧挨着的雇员记录。

- 为此对 match 查询稍作调整，使用一个叫做 match_phrase 的查询：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
```

```
    }
}
```

- 毫无悬念，返回结果仅有 John Smith 的文档。

```
{
  ...
  "hits": {
    "total":      1,
    "max_score":  0.23013961,
    "hits": [
      {
        ...
        "_score":         0.23013961,
        "_source": {
          "first_name":  "John",
          "last_name":   "Smith",
          "age":         25,
          "about":       "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        }
      }
    ]
  }
}
```

- 许多应用都倾向于在每个搜索结果中 *高亮* 部分文本片段，以便让用户知道为何该文档符合查询条件。在 Elasticsearch 中检索出高亮片段也很容易。
  - 再次执行前面的查询，并增加一个新的 highlight 参数：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    },
    "highlight": {
        "fields" : {
            "about" : {}
        }
    }
}
```

  - 当执行该查询时，返回结果与之前一样，与此同时结果中还多了一个叫做 highlight 的部分。这个部分包含了 about 属性匹配的文本片段，并以 HTML 标签 封装：

```
{
  ...
  "hits": {
    "total":      1,
    "max_score":  0.23013961,
    "hits": [
      {
        ...
        "_score":         0.23013961,
        "_source": {
          "first_name":  "John",
          "last_name":   "Smith",
          "age":         25,
          "about":       "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        },
        "highlight": {
          "about": [
            "I love to go <em>rock</em> <em>climbing</em>"
          ]
        }
      }
    ]
  }
}
```

- 5、spring-boot整合elasticsearch
  - 1、pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
```

  - 2、application.properties

```
spring.elasticsearch.rest.uris=http://39.101.170.233:9200
```

  - 3、编写实体类带上注释@Document(indexName = "atguigu")

```
package com.lhq.elastic.bean;
import org.springframework.data.elasticsearch.annotations.Document;
@Document(indexName = "atguigu")
public class Book {
    private Integer id;
    private String bookName;
    private String author;
```

```
    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", bookName='" + bookName + '\'' +
                ", author='" + author + '\'' +
                '}';
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getBookName() {
        return bookName;
    }
    public void setBookName(String bookName) {
        this.bookName = bookName;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}
```

- 4、写一个repository继承ElasticsearchRepository

```
package com.lhq.elastic.repository;
import com.lhq.elastic.bean.Book;
import
org.springframework.data.elasticsearch.repository.ElasticsearchRepository;
public interface BookRepository extends
ElasticsearchRepository<Book,Integer> {
}
```

- 5、测试
  - 1）存入

```
package com.lhq.elastic;
import com.lhq.elastic.bean.Book;
import com.lhq.elastic.repository.BookRepository;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
class Springboot03elasticApplicationTests {
    @Autowired
    BookRepository bookRepository;
    @Test
    void contextLoads() {
        Book book = new Book();
        book.setBookName("西游记");
        book.setId(1);
        book.setAuthor("吴承恩");
        Book save = bookRepository.save(book);
        System.out.println(save);
    }
}
```

- 2）请求

  http://39.101.170.233:9200/atguigu/book/1

- 3）自定义查询

```
package com.lhq.elastic.repository;
import com.lhq.elastic.bean.Book;
import
org.springframework.data.elasticsearch.repository.ElasticsearchReposi
tory;

import java.util.List;

public interface BookRepository extends
ElasticsearchRepository<Book,Integer> {
    public List<Book> findBookByBookNameLike(String name);
}
```