
Table of Contents

写在前面的话	1.1
准备工作	1.2
基本术语	1.3
简单API	1.4
分词	1.5
简单搜索	1.6
Java客户端（上）	1.7
父-子关系文档	1.8
复杂搜索	1.9
Java客户端（下）	1.10
实战：ELK日志分析系统	1.11
实战：多数据源同步	1.12

写在前面的话

[查看教程](#)

这个教程虽然名字叫做《ElasticSearch6.x实战教程》但实际上离“实战”，离“教程”还相去甚远。原本是打算基于ElasticSearch5.x，但写到“父-子关系文档”一章时发现5.x与6.x在父子文档关系上完全不同，其根本原因在于ElasticSearch正在逐步放弃一个索引（Index）建立多个类型（Type）。从ElasticSearch6.x开始，官方只允许一个索引（Index）建立一个类型（Type）。在未来的版本中，类型（Type）这一概念将会被完全移除。在询问身边的朋友所在公司使用的ElasticSearch版本大多是基于ElasticSearch6.x后，决定将基于ElasticSearch6.x开始ElasticSearch之旅。

说回离实战和教程相去甚远，第一，这个“教程”并不完美，并没有将ElasticSearch的魅力充分展现，仅仅展露了其“冰山一角”。第二，这个“教程”也并不深入，既没有“由浅入深”的层层递进，也没有“从入门到精通”的强大魄力。第三，说到“实战”，也没有引人入胜的“BAT大厂ES面经”，最后的实战章节也只能管中窥豹。

说了我自己写的那么多不好，是不是就立马放弃这个“教程”了呢？我想，把这个“教程”当做是学习ElasticSearch的第一步是没有错的。尽管有那么多的“缺点”，但这个“教程”我尽可能把每个执行的HTTP请求参数完整的记录下来，尽可能的思考可能遇到的实际场景，例如在“复杂搜索”章节中的“搜索精度”问题，在实际生活当中我们在搜索一个关键字时，给出关键字往往并不那么准确，但我们却希望搜索结果能准确。我们搜索“新希望牛奶”或者“牛奶新希望”时，绝不希望出现“春夏上新短袖”这一结果。

如果看到这个所谓的“教程”，不妨读下去，也许它就是你的ES启蒙小册。

OKevin

2019年7月9日凌晨 于成都

准备工作

工欲善其事必先利其器

ElasticSearch安装

ElasticSearch6.3.2下载地址（Linux、mac OS、Windows通用，下载zip包即可）：<https://www.elastic.co/cn/downloads/past-releases/elasticsearch-6-3-2>。ES历史版本下载页面：<https://www.elastic.co/cn/downloads/past-releases#elasticsearch>。

在正式安装前，你需要确保你的系统已配置JDK8环境。

mac OS

在上述下载地址下载完elasticsearch-6.3.2.tar.gz后，首先在当前登录用户的 home 下创建一个 Settings 目录，通过 tar -zxvf elasticsearch-6.3.2.tar.gz 解压到当前目录。

进入 elasticsearch-6.3.2.tar.gz 目录，执行 ./bin/elasticsearch 命令，等待一小段时间，通过浏览器访问 <http://localhost:9200/?pretty> 出现以下响应：

```
{
  "name": "x4x7wWJ",
  "cluster_name": "elasticsearch",
  "cluster_uuid": "sJ6LTYJ1TDmtR1kzl0M2Ig",
  "version": {
    "number": "6.3.2",
    "build_hash": "8bbedf5",
    "build_date": "2017-10-31T18:55:38.105Z",
    "build_snapshot": false,
    "lucene_version": "6.6.1"
  },
  "tagline": "You Know, for Search"
}
```

Linux

Linux的安装过程和Linux相同。

ES需要使用普通用户安装、启动，如果你是root用户，需要先创建一个用户，用普通用户而不是root用户启动ES。

基本术语

白马非马

ES是一个搜索引擎，同时它也是一个分布式文档存储数据库（当然是非关系型的）。为了保证后续的实战教程顺利进行，这里通过对传统的关系型数据库MySQL介绍在ES中的一些术语。

在MySQL中共有数据库（Database）、表（Table）、记录（Row）、列（Column）的概念，同样在ES中也有类似的概念，索引（Index），类型（Type），文档（Document），字段（Field）。

可以这么理解：

	数据库	表	记录	列
MySQL	DB	Table	Row	Column
ES	Index	Document	Document	Field

索引|Index

ES中的索引概念可不是关系型数据库中的“索引”，ES中的索引指的是存储数据的地方，类似关系型数据库中的数据库概念。

类型Type

有的文章指出ES中的类型Type对应的就是关系型数据库中的表，在使用ES中我们会遇到另外一个概念映射（Mapping），也有不少的文章指出Mapping对应的就是关系型数据库中的表。关系型数据库中表与表是物理独立的，即使在两个表中存在相同名称不同类型的列，这在我们的关系型数据库也是极为合理的，但这在ES中就不合理，ES中即使是在同一个索引Index下，如果字段Field存在于不同的类型Type中，即使他们代表不同的含义，但是只要它们的名称相同也必须要求类型相同，在ES中类型Type对应于关系型数据库中表的概念已经名存实亡。实际上在ES中Type作为表的概念在后期版本中越来越被弱化，在未被ES正式移除前，ES后期版本已经不允许一个索引Index创建多个Type，相信在后面的版本会彻底移除类型Type。

（注：ES6已经不允许一个Index创建多个Type，<https://github.com/elastic/elasticsearch/pull/24317>）

如果在现阶段一定要理解ES中的Type，那么一定要和Mapping结合起来。可以理解为类型Type就是定义一个表，仅仅是定义而已，而映射Mapping定义了表结构（有哪些列，列的类型是什么）。

文档Document

在非关系型数据库中，有部分被称之为“文档数据库”，对应于关系型数据库中的一行记录。

字段Field

对应关系型数据库中的列。

节点

一个ES实例称之为一个节点，单机部署的ES有且只有一个节点，集群部署的ES有多个节点且有一个主节点。

分片

ES可作为分布式集群部署，同样也可以作为单机单节点部署。ES中的数据被分散存储在分片中，ES屏蔽了底层的分片实现，我们直接与索引交互而不与分片交互。分片数量的多少与是否是集群部署和单机部署无关，即使是单机部署在创建索引时仍然也可以指定划分多个分片（默认5个主分片1份备份（包含5个备分片））。分片有主分片和备分片之分，顾名思义，备分片是主分片的备份，当主分片出现故障时，备份片充当主分片。

对于单机部署

单机部署的ES，即表示ES有且只有一个节点，在创建索引时，如果不指定主分片与备分片的数量，默然创建5个主分片和1份备份（5个备分片），实际上对于单机部署的ES服务来讲，多个主分片并没有意义，多个分片存在的意义本身就是将数据分散存储到多个ES节点中进行同时查询，此时只有一个节点多个分片也没有意义。备分片在单机部署中同样也没有意义，备份存在的意义本身就是当主分片故障时，仍然能对外提供服务，此时主备都在一个节点上，如果主分片故障，备分片也同样会导致故障。

对于集群部署

对于集群部署的ES来讲，此时存在多个节点，主分片的分配与备分片机制就显得尤为重要（这涉及查询性能以及服务高可用），例如现在有3个节点，此时如果在创建索引时只分配1个主分片就显得有点浪费（注：主分片一旦在创建索引时确定便不能修改）。主分片的划分并没有一定放之四海而皆准的规则，更多的是取决于用户的数据量以及ES节点数量等。通常所理解的是，分片数量越多越好，因为这能将数据分散到不同分片，以便以后在扩容新增节点时，ES能将自动将分片重新进行均匀分布。但这条理论也不绝对，如果你的节点只有3个，设置了100个分片，每个节点就有33个节点，当搜索请求调度到同一节点的不同分片时，此时会引发硬件资源（CPU）的抢夺，造成性能问题。反过来，如果3个节点只分配了3个分片，随着业务的发展，数据量越来越大，单个分片已不能承受它最大的数据量，此时就算新增节点，但是分片数量只有3个，分片的数量在创建索引时便确定且不可修改，此时只能通过重新创建索引。

既要对合理的数据增长有一个判断（规划较大的分片），又要对期望有一个度的把握（合理的分片数量）。官方给出了一点建议，每个分片的数据量最好是在20G~40G左右，这就意味着如果你有4个节点，数据量预估在200G左右甚至更大，此时分片数量设置为5~10个比较合适，7、8个差不多，每个节点有2个分片。（官方博客的建议，<https://www.elastic.co/cn/blog/how-many-shards-should-i-have-in-my-elasticsearch-cluster>）

上面谈到的是主分片，副分片的划分也同等重要。如果不对分片备份，主分片故障则导致数据丢失，部分数据不可查询。副本分片设置过多造成额外的存储空间，默然情况下，创建索引时会创建一个分片副本（一个分片副本不代表一个备分片，如果有5个主分片，那么分片副本就有5个备分片，同理如果指定创建两个分片副本，此时一共就有10个备分片。）需要注意的是备分片是可以修改的，所以备分片可以直接采用默然一个分片副本。

简单的API

万丈高楼平地起

ES提供了多种操作数据的方式，其中较为常见的方式就是RESTful风格的API。

简单的体验

利用Postman发起HTTP请求（当然也可以在命令行中使用curl命令）。

索引|Index

创建索引

创建一个名叫 demo 的索引：

```
PUT http://localhost:9200/demo
```

ES响应：

```
{  
    "acknowledged": true,  
    "shards_acknowledged": true,  
    "index": "demo"  
}
```

在创建索引时，可指定主分片和分片副本的数量：

```
PUT http://localhost:9200/demo
```

```
{  
    "settings":{  
        "number_of_shards":1,  
        "number_of_replicas":1  
    }  
}
```

ES响应：

```
{  
    "acknowledged": true,  
    "shards_acknowledged": true,  
    "index": "demo"  
}
```

查看指定索引

```
GET http://localhost:9200/demo
```

ES响应：

```
{
  "demo": {
    "aliases": {},
    "mappings": {},
    "settings": {
      "index": {
        "creation_date": "1561110747038",
        "number_of_shards": "1",
        "number_of_replicas": "1",
        "uuid": "kjPqDUT6TMyywg1P7qgccw",
        "version": {
          "created": "5060499"
        },
        "provided_name": "demo"
      }
    }
  }
}
```

查询ES中的索引

查询ES中索引情况：

```
GET http://localhost:9200/_cat/indices?v
```

ES响应：

health	status	index	uuid	pri	rep	docs.count
yellow	open	demo	wqkto5CCTpWNdP3HGpLfxA	5	1	0
yellow	open	.kibana	pwKW9hJyRkO7_pE0MNE05g	1	1	1

可以看到当前ES中一共有2个索引，一个是我们刚创建的 `demo`，另一个是kibana创建的索引 `.kibana`。表格中有一些信息代表了索引的一些状态。

health: 健康状态，`red`表示不是所有的主分片都可用，即部分主分片可用。`yellow`表示主分片可用备分片不可用，常常是单机ES的健康状态，`green`表示所有的主分片和备分片都可用。（官方对集群健康状态的说明，<https://www.elastic.co/guide/en/elasticsearch/guide/master/cluster-health.html>）

status: 索引状态，`open`表示打开可对索引中的文档数据进行读写，`close`表示关闭此时索引占用的内存会被释放，但是此时索引不可进行读写操作。

index: 索引

uuid: 索引标识

pri: 索引的主分片数量

rep: 索引的分片副本数量, 1表示有一个分片副本 (有多少主分片就有多少备分片, 此处表示5个备分片)。

docs.count: 文档数量

docs.deleted: 被删除的文档数量

store.size: 索引大小

pri.store.size: 主分片占用的大小

删除索引

删除 demo 索引, 删除索引等同于删库跑路, 请谨慎操作。

```
DELETE http://localhost:9200/demo
```

ES响应:

```
{
  "acknowledged": true
}
```

类型Type (同时定义映射Mapping字段及类型)

创建类型

在前面基本术语中我们提到类型Type类似关系型数据库中的表, 映射Mapping定义表结构。创建类型Type时需要配合映射Mapping。

创建索引 demo 的类型为 example_type ,包含两个字段: created 类型为date, message 类型为 keyword:

方式一:

```
PUT http://localhost:9200/demo/_mapping/example_type
```

```
{
  "properties": {
    "created": {
      "type": "date"
    },
    "message": {
      "type": "keyword"
    }
  }
}
```

此时再次执行查询索引的操作，已经可以发现类型Type被创建了，遗憾的是，如果类型Type（或者映射Mapping）一旦定义，就不能删除，只能修改，为了保证本教程顺利进行方式二创建类型，所以此处执行 `DELETE http://localhost:9200/demo` 删除索引。删除索引后不要再创建索引，下面的这种方式是在创建索引的同时创建Type并定义Mapping

方式二：

```
PUT http://localhost:9200/demo
{
  "mappings": {
    "example_type": {
      "properties": {
        "created": {
          "type": "date"
        },
        "message": {
          "type": "keyword"
        }
      }
    }
  }
}
```

此时执行 `GET http://localhost:9200/demo`，可以看到我们在ES中创建了第一个索引以及创建的表结构，接下来插入就是数据（即文档）。

文档Document

插入文档

系统定义 `_id`

```
POST http://localhost:9200/demo/example_type
{
  "created": 1561135459000,
  "message": "test1"
}
```

ES响应：

```
{
  "_index": "demo",
  "_type": "example_type",
  "_id": "AWt67Q1_Tf0FgxupY1BX",
  "_version": 1,
  "result": "created",
  "_shards": {
```

```

    "total": 2,
    "successful": 1,
    "failed": 0
},
"created": true
}

```

查询文档

ElasticSearch的核心功能——搜索。

```
POST http://localhost:9200/demo/example_type/_search?pretty
```

ES响应：

```
{
  "took": 183,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1,
    "hits": [
      {
        "_index": "demo",
        "_type": "example_type",
        "_id": "AWt67Ql_Tf0FgxupY1BX",
        "_score": 1,
        "_source": {
          "created": 1561135459000,
          "message": "test1"
        }
      }
    ]
  }
}
```

关于文档的查询是ElasticSearch的核心，后面的章节会详细介绍一些基本的简单查询和更为高级的复杂查询，此处仅作为对插入数据的验证，不做过多展开。

修改文档

根据文档 `_id` 修改

```
POST http://localhost:9200/demo/example_type/AWt67Ql_Tf0FgxupYlBX/_update
{
  "doc": {
    "message": "updated"
  }
}
```

ES响应：

```
{
  "_index": "demo",
  "_type": "example_type",
  "_id": "AWt67Ql_Tf0FgxupYlBX",
  "_version": 2,
  "result": "updated",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  }
}
```

删除文档

删除 _id 为 AWt67Ql_Tf0FgxupYlBX 的文档

```
DELETE http://localhost:9200/demo/example_type/AWt67Ql_Tf0FgxupYlBX
```

ES的响应：

```
{
  "found": true,
  "_index": "demo",
  "_type": "example_type",
  "_id": "AWt67Ql_Tf0FgxupYlBX",
  "_version": 2,
  "result": "deleted",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  }
}
```


分词

下雨天留客天留我不留

本打算先介绍“简单搜索”，对ES的搜索有一个直观的感受。但在写的过程中发现分词无论如何都绕不过去。`term` 查询，`match` 查询都与分词息息相关，索性先介绍分词。

ES作为一个开源的搜索引擎，其核心自然在于搜索，而搜索不同于我们在MySQL中的 `select` 查询语句，无论我们在百度搜索一个关键字，或者在京东搜索一个商品时，常常无法很准确的给出一个关键字，例如我们在百度希望搜索“Java教程”，我们希望结果是“Java教程”、“Java”、“Java基础教程”，甚至是“教程Java”。MySQL虽然能满足前三种查询结果，但却无法满足最后一种搜索结果。

虽然我们很难做到对于百度或者京东的搜索（这甚至需要了解Lucene和搜索的底层原理），但我们能借助ES做出一款不错的搜索产品。

ES的搜索中，分词是非常重要的概念。掌握分词原理，对待一个不甚满意的搜索结果我们能定位是哪里出了问题，从而做出相应的调整。

ES中，只对字符串进行分词，在ElasticSearch2.x版本中，字符串类型只有 `string`，ElasticSearch5.x版本后字符串类型分为了 `text` 和 `keyword` 类型，需要明确的分词只在 `text` 类型。

ES的默认分词器是 `standard`，对于英文搜索它没有问题，但对于中文搜索它会将所有的中文字符串挨个拆分，也就是它会将“中国”拆分为“中”和“国”两个单词，这带来的问题是搜索关键字为“中国”时，将不会有任何结果，ES会将搜索字段进行拆分后搜索。当然，你可以指定让搜索的字段不进行分词，例如设置为 `keyword` 字段。

分词体验

前面说到ES的默认分词器是 `standard`，可直接通过API指定分词器以及字符串查看分词结果。

使用 `standard` 进行英文分词：

```
POST http://localhost:9200/_analyze
{
  "analyzer": "standard",
  "text": "hello world"
}
```

ES响应：

```
{
  "tokens": [
    {
      "token": "hello",
      "start_offset": 0,
```

```

        "end_offset": 5,
        "type": "<ALPHANUM>",
        "position": 0
    },
    {
        "token": "world",
        "start_offset": 6,
        "end_offset": 11,
        "type": "<ALPHANUM>",
        "position": 1
    }
]
}

```

如果我们对“helloworld”进行分词，结果将只有“helloworld”一个词，`standard` 对英文按照空格进行分词。

使用 `standard` 进行中文分词：

```

POST http://localhost:9200/_analyze
{
    "analyzer": "standard",
    "text": "学生"
}

```

ES响应：

```

{
    "tokens": [
        {
            "token": "学",
            "start_offset": 0,
            "end_offset": 1,
            "type": "<IDEOGRAPHIC>",
            "position": 0
        },
        {
            "token": "生",
            "start_offset": 1,
            "end_offset": 2,
            "type": "<IDEOGRAPHIC>",
            "position": 1
        }
    ]
}

```

“学生”显然应该是一个词，不应该被拆分。也就是说如果字符串中是中文，默认的 `standard` 不符合我们的需求。幸运地是，ES支持第三方分词插件。在ES中的中文分词插件使用最为广泛的是ik插件。

ik插件

既然是插件，就需要安装。注意，版本5.0.0起，ik插件已经不包含名为 `ik` 的分词器，只含 `ik_smart` 和 `ik_max_word`，事实上后两者使用得也最多。

ik插件安装

ik下载地址（直接下载编译好了的zip文件，需要和ES版本一致）：<https://github.com/medcl/elasticsearch-analysis-ik/releases/tag/v6.3.2>。ik历史版本下载页面：<https://github.com/medcl/elasticsearch-analysis-ik/releases>。

下载完成后解压 `elasticsearch-analysis-ik-6.3.2.zip` 将解压后的文件夹直接放入ES安装目录下的 `plugins` 文件夹中，重启ES。

使用ik插件的 `ik_smart` 分词器：

```
POST http://localhost:9200/_analyze
{
  "analyzer": "ik_smart",
  "text": "学生"
}
```

ES响应：

```
{
  "tokens": [
    {
      "token": "学生",
      "start_offset": 0,
      "end_offset": 2,
      "type": "CN_WORD",
      "position": 0
    }
  ]
}
```

这才符合我们的预期。那么ik插件中的 `ik_smart` 和 `ik_max_word` 有什么区别呢？简单来讲，`ik_smart` 会按照关键字的最粗粒度进行分词，比如搜索“北京大学”时，我们知道“北京大学”是一个特定的词汇，它并不是指“北京的大学”，我们不希望搜索出“四川大学”，“重庆大学”等其他学校，此时“北京大学”不会被分词。而 `ik_max_word` 则会按照最细粒度进行分词，同样搜索“北京大学”时，我们也知道“北京”和“大学”都是一个词汇，所以它将会被分词为“北京大学”，“北京大”，“北京”，“大学”，显然如果搜索出现后三者相关结果，这会给我们带来更多无用的信息。

所以我们在进行搜索时，常常指定 `ik_smart` 为分词器。

有时候一个词并不在ik插件的词库中，例如很多网络用语等。我们希望搜索“小米手机”的时候，只出现“小米的手机”而不会出现“华为手机”、“OPPO手机”，但“小米手机”并不在ik词库中，此时我们可以将“小米手机”添加到ik插件的自定义词库中。

“小米手机”使用 `ik_smart` 的分词结果：

```
{
  "tokens": [
    {
      "token": "小米",
      "start_offset": 0,
      "end_offset": 2,
      "type": "CN_WORD",
      "position": 0
    },
    {
      "token": "手机",
      "start_offset": 2,
      "end_offset": 4,
      "type": "CN_WORD",
      "position": 1
    }
  ]
}
```

进入ik插件安装目录 `elasticsearch-5.6.0/plugins/elasticsearch/config`，创建名为 `custom.dic` 的自定义词库，向文件中添加“小米手机”并保存。仍然是此目录，修改 `IKAnalyzer.cfg.xml` 文件，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>IK Analyzer 扩展配置</comment>
  <!--用户可以在这里配置自己的扩展字典-->
  <entry key="ext_dict">custom.dic</entry>
  <!--用户可以在这里配置自己的扩展停止词字典-->
  <entry key="ext_stopwords"></entry>
  <!--用户可以在这里配置远程扩展字典-->
  <!-- <entry key="remote_ext_dict">words_location</entry> -->
  <!-- 用户可以在这里配置远程扩展停止词字典-->
  <!-- <entry key="remote_ext_stopwords">words_location</entry> -->
</properties>
```

重启ES后，再次通过 `ik_smart` 对“小米手机”进行分词，发现“小米手机”不再被分词。

创建映射指定分词器

在创建映射时，我们可以指定字段采用哪种分词器，避免我们在每次搜索时都指定。

1. 创建word索引 `PUT http://localhost:9200/word`

2. 创建analyzer_demo类型已经定义映射Mapping

```
PUT http://localhost:9200/word/analyzer_demo/_mapping
{
  "properties": {
    "name": {
      "type": "text",
      "analyzer": "ik_smart"
    }
  }
}
```

3. 查看word索引结构 `GET http://localhost:9200/word`

ES响应:

```
{
  "word": {
    "aliases": {},
    "mappings": {
      "analyzer_demo": {
        "properties": {
          "name": {
            "type": "text",
            "analyzer": "ik_smart"
          }
        }
      }
    },
    "settings": {
      "index": {
        "creation_date": "1561304920088",
        "number_of_shards": "5",
        "number_of_replicas": "1",
        "uuid": "A2Y09GpzRrGAIm2Q6rCoWA",
        "version": {
          "created": "5060099"
        },
        "provided_name": "word"
      }
    }
  }
}
```

可以看到ES在对name字段进行分词时会采用 `ik_smart` 分词器。

简单搜索

众里寻他千百度

搜索是ES的核心，本节讲解一些基本的简单的搜索。

掌握ES搜索查询的RESTful的API犹如掌握关系型数据库的SQL语句，尽管Java客户端API为我们不需要我们去实际编写RESTful的API，但在生产环境中，免不了在线上执行查询语句做数据统计供产品经理等使用。

数据准备

首先创建一个名为user的Index，并创建一个student的Type，Mapping映射一共有如下几个字段：

1. 创建名为user的Index `PUT http://localhost:9200/user`
2. 创建名为student的Type，且指定字段name和address的分词器为 `ik_smart`。

```
POST http://localhost:9200/user/student/_mapping
{
  "properties": {
    "name": {
      "type": "text",
      "analyzer": "ik_smart"
    },
    "age": {
      "type": "short"
    }
  }
}
```

经过上一章分词的学习我们把 `text` 类型都指定为 `ik_smart` 分词器。

插入以下数据。

```
POST localhost:9200/user/student
{
  "name": "kevin",
  "age": 25
}
```

```
POST localhost:9200/user/student
{
  "name": "kangkang",
  "age": 26
}
```

```
POST localhost:9200/user/student
{
  "name": "mike",
  "age": 22
}
```

```
POST localhost:9200/user/student
{
  "name": "kevin2",
  "age": 25
}
```

```
POST localhost:9200/user/student
{
  "name": "kevin yu",
  "age": 21
}
```

按查询条件数量维度

无条件搜索

```
GET http://localhost:9200/user/student/_search?pretty
```

查看索引user的student类型数据，得到刚刚插入的数据返回：

单条件搜索

ES查询主要分为 `term` 精确搜索、`match` 模糊搜索。

term精确搜索

我们用 `term` 搜索name为“kevin”的数据。

```
POST http://localhost:9200/user/student/_search?pretty
{
  "query": {
    "term": {
      "name": "kevin"
    }
  }
}
```

既然 `term` 是精确搜索，按照非关系型数据库的理解来讲就等同于 `=`，那么搜索结果也应该只包含1条数据。然而出乎意料的是，搜索结果出现了两条数据：`name="kevin"`和`name="keivin yu"`，这看起来似乎是进行的模糊搜索，但又没有搜索出`name="kevin2"`的数据。我们先继续观察 `match` 的搜索结果。

match模糊搜索

同样，搜索`name`为“`kevin`”的数据。

```
POST http://localhost:9200/user/student/_search?pretty
{
  "query": {
    "match": {
      "name": "kevin"
    }
  }
}
```

`match` 的搜索结果竟然仍然是两条数据：`name="kevin"`和`name="keivin yu"`。同样，`name="kevin2"`也没有出现在搜索结果中。

原因在于 `term` 和 `match` 的精确和模糊针对的是搜索词而言，`term` 搜索不会将搜索词进行分词后再搜索，而 `match` 则会将搜索词进行分词后再搜索。例如，我们对`name="kevin yu"`进行搜索，由于 `term` 搜索不会对搜索词进行搜索，所以它进行检索的是“`kevin yu`”这个整体，而 `match` 搜索则会对搜索词进行分词搜索，所以它进行检索的是包含“`kevin`”和“`yu`”的数据。而`name`字段是 `text` 类型，且它是按照 `ik_smart` 进行分词，就算是“`kevin yu`”这条数据由于被分词后变成了“`kevin`”和“`yu`”，所以 `term` 搜索不到任何结果。

如果一定要用 `term` 搜索`name="kevin yu"`，结果出现“`kevin yu`”，办法就是在定义映射Mapping时就为该字段设置一个 `keyword` 类型。

为了下文的顺利进行，删除 `DELETE http://localhost:9200/user/student` 重新按照开头创建索引以及插入数据吧。唯一需要修改的是在定义映射Mapping时，`name`字段修改为如下所示：

```
{
  "properties": {
    "name": {
      "type": "text",
      "analyzer": "ik_smart",
      "fields": {
        "keyword": {
          "type": "keyword",
          "ignore_above": 256
        }
      }
    },
    "age": {
      "type": "integer"
    }
}
```

```

    }
}
}
```

待我们重新创建好索引并插入数据后，此时再按照 `term` 搜索`name="kevin yu"`。

```
POST http://localhost:9200/user/student/_search
{
  "query": {
    "term": {
      "name.keyword": "kevin yu"
    }
  }
}
```

返回一条`name="kevin yu"`的数据。按照 `match` 搜索同样出现`name="kevin yu"`，因为`name.keyword`无论如何都不会再分词。

在已经建立索引且定义好映射*Mapping*的情况下，如果直接修改`name`字段，此时能修改成功，但是却无法进行查询，这与ES底层实现有关，如果一定要修改要么是新增字段，要么是重建索引。

所以，与其说 `match` 是模糊搜索，倒不如说它是分词搜索，因为它会将搜索关键字分词；与其将 `term` 称之为模糊搜索，倒不如称之为不分词搜索，因为它不会将搜索关键字分词。

```
##### 类似like的模糊搜索

```wildcard```通配符查询。

```json
POST http://localhost:9200/user/student/_search?pretty
{
  "query": {
    "wildcard": {
      "name": "*kevin*"
    }
  }
}
```

ES返回结果包括`name="kevin"`, `name="kevin2"`, `name="kevin yu"`。

fuzzy更智能的模糊搜索

`fuzzy`也是一个模糊查询，它看起来更加“智能”。它类似于搜狗输入法中允许语法错误，但仍能搜出你想要的结果。例如，我们查询`name`等于“kevin”的文档时，不小心输成了“kevon”，它仍然能查询出结构。

```
POST http://localhost:9200/user/student/_search?pretty
```

```
{
  "query": {
    "fuzzy": {
      "name": "kevin"
    }
  }
}
```

ES返回结果包括name="kevin"， name="kevin yu"。

多条件搜索

上文介绍了单个条件下的简单搜索，并且介绍了相关的精确和模糊搜索（分词与不分词）。这部分将介绍多个条件下的简单搜索。

当搜索需要多个条件时，条件与条件之间的关系有“与”，“或”，“非”，正如非关系型数据库中的“and”，“or”，“not”。

在ES中表示“与”关系的是关键字 `must`，表示“或”关系的是关键字 `should`，还有表示“非”的关键字 `must_not`。

1. 精确查询（`term`，搜索关键字不分词）name="kevin"且age="25"的学生。

```
```json
POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
 "bool": {
 "must": [
 {
 "term": {
 "name.keyword": "kevin"
 }
 },
 {
 "term": {
 "age": 25
 }
 }
]
 }
 }
}
```

返回name="kevin"且age="25"的数据。

1. 精确查询（`term`，搜索关键字不分词）name="kevin"或age="21"的学生。

```
POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
```

```

"bool": {
 "should": [
 {
 "term": {
 "name.keyword": "kevin"
 }
 },
 {
 "term": {
 "age": 21
 }
 }
]
}
}

```

返回name="kevin"， age=25和name="kevin yu"， age=21的数据

1. 精确查询 ( term ， 搜索关键字不分词) name!="kevin"且age="25"的学生。

```

POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
 "bool": {
 "must": [
 {
 "term": {
 "age": 25
 }
 }
],
 "must_not": [
 {
 "term": {
 "name.keyword": "kevin"
 }
 }
]
 }
 }
}

```

返回name="kevin2"的数据。

如果查询条件中同时包含 must 、 should 、 must\_not ， 那么它们三者是"且"的关系

多条件查询中查询逻辑( must 、 should 、 must\_not )与查询精度( term 、 match )配合能组合成非常丰富的查询条件。

## 按等值、范围查询维度

上文中讲到了精确查询、模糊查询，已经"且"，"或"，"非"的查询。基本上都是在做等值查询，实际查询中还包括，范围（大于小于）查询 ( range ) 、存在查询 ( exists ) 、~不存在查询 (~missing)。

## 范围查询

范围查询关键字 `range`，它包括大于 `gt`、大于等于 `gte`、小于 `lt`、小于等于 `lte`。

1. 查询`age>25`的学生。

```
POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
 "range": {
 "age": {
 "gt": 25
 }
 }
 }
}
```

返回`name="kangkang"`的数据。

1. 查询`age >= 21`且`age < 26`的学生。

```
POST http://localhost:9200/user/search/_search?pretty
{
 "query": {
 "range": {
 "age": {
 "gte": 21,
 "lt": 25
 }
 }
 }
}
```

查询`age >= 21`且`age < 26`且`name="kevin"`的学生

```
POST http://localhost:9200/user/search/_search?pretty
{
 "query": {
 "bool": {
 "must": [
 {
 "term": {
 "name": "kevin"
 }
 },
 {
 "range": {
 "age": {
 "gte": 21,
 "lt": 25
 }
 }
 }
]
 }
 }
}
```

```

 }
 }]
}
}
}
```

## 存在查询

存在查询意为查询是否存在某个字段。

查询存在name字段的数据。

```
POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
 "exists": {
 "field": "name"
 }
 }
}
```

## 不存在查询

不存在查询顾名思义查询不存在某个字段的数据。在以前ES有 missing 表示查询不存在的字段，后来的版本中由于 must not 和 exists 可以组合成 missing，故去掉了 missing。

查询不存在name字段的数据。

```
POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
 "bool": {
 "must_not": {
 "exists": {
 "field": "name"
 }
 }
 }
 }
}
```

## 分页搜索

谈到ES的分页永远都绕不开深分页的问题。但在本章中暂时避开这个问题，只说明在ES中如何进行分页查询。

ES分页查询包含 from 和 size 关键字，from 表示起始值，size 表示一次查询的数量。

## 1. 查询数据的总数

```
POST http://localhost:9200/user/student/_search?pretty
```

返回文档总数。

### 1. 分页（一页包含1条数据）模糊查询( `match` , 搜索关键字不分词)name="kevin"

```
POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
 "match": {
 "name": "kevin"
 }
 },
 "from": 0,
 "size": 1
}
```

结合文档总数即可返回简单的分页查询。

分页查询中往往我们也需要对数据进行排序返回，MySQL中使用 `order by` 关键字，ES中使用 `sort` 关键字指定排序字段以及降序升序。

### 1. 分页（一页包含1条数据）查询`age >= 21`且`age <= 26`的学生，按年龄降序排列。

```
POST http://localhost:9200/user/student/_search?pretty
{
 "query": {
 "range": {
 "age": {
 "gte": 21,
 "lte": 26
 }
 }
 },
 "from": 0,
 "size": 1,
 "sort": {
 "age": {
 "order": "desc"
 }
 }
}
```

ES默认升序排列，如果不指定排序字段的排序），则 `sort` 字段可直接写为 `"sort": "age"` 。



## Java客户端（上）

ES提供了多种方式使用Java客户端：

- TransportClient，通过Socket方式连接ES集群，传输会对Java进行序列化
- RestClient，通过HTTP方式请求ES集群

目前常用的是 TransportClient 方式连接ES服务。但ES官方表示，在未来 TransportClient 会被永久移除，只保留 RestClient 方式。

同样，Spring Boot官方也提供了操作ES的方式 Spring Data Elasticsearch 。本章节将首先介绍基于Spring Boot所构建的工程通过 Spring Data Elasticsearch 操作ES，再介绍同样是基于Spring Boot所构建的工程，但使用ES提供的 TransportClient 操作ES。

## Spring Data Elasticsearch

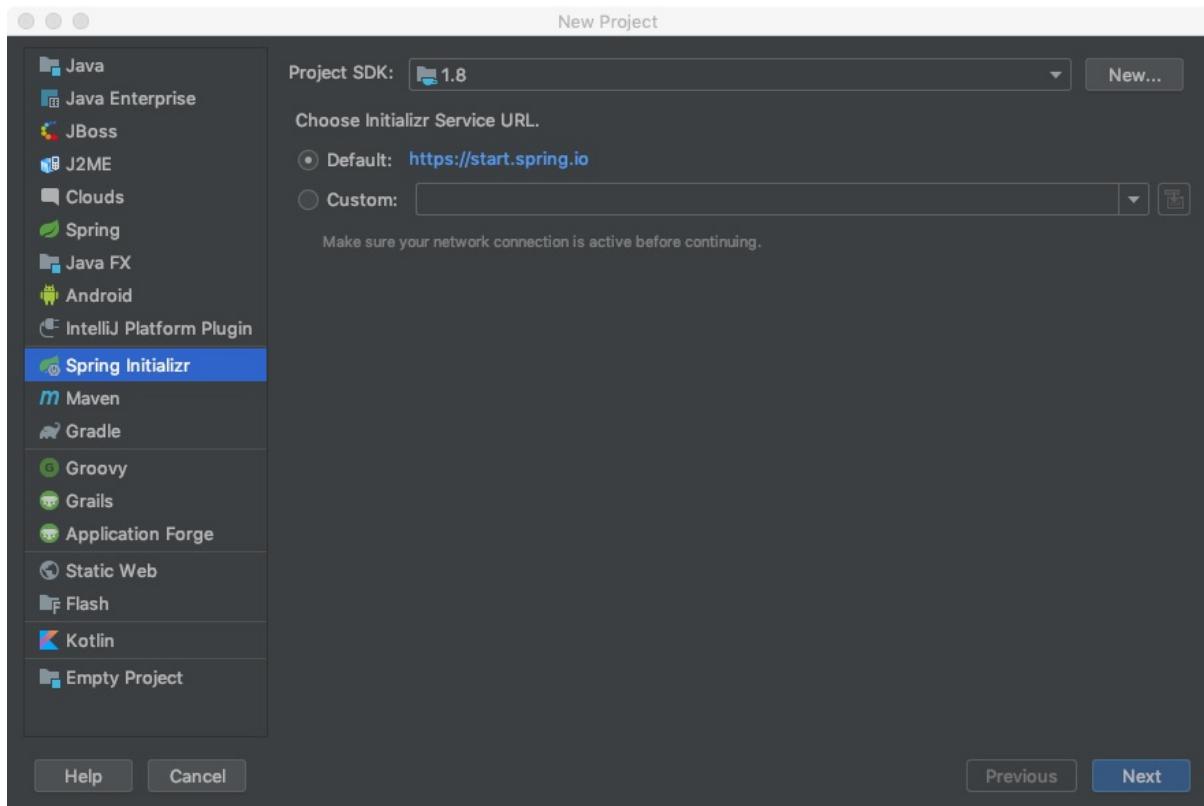
本节完整代码（配合源码使用更香）：[https://github.com/yulinfeng/elasticsearch6.x\\_tutorial/tree/master/code/spring-data-elasticsearch](https://github.com/yulinfeng/elasticsearch6.x_tutorial/tree/master/code/spring-data-elasticsearch)

使用 Spring Data Elasticsearch 后，你会发现一切变得如此简单。就连连接ES服务的类都不需要写，只需要配置一条ES服务在哪儿的信息就能开箱即用。

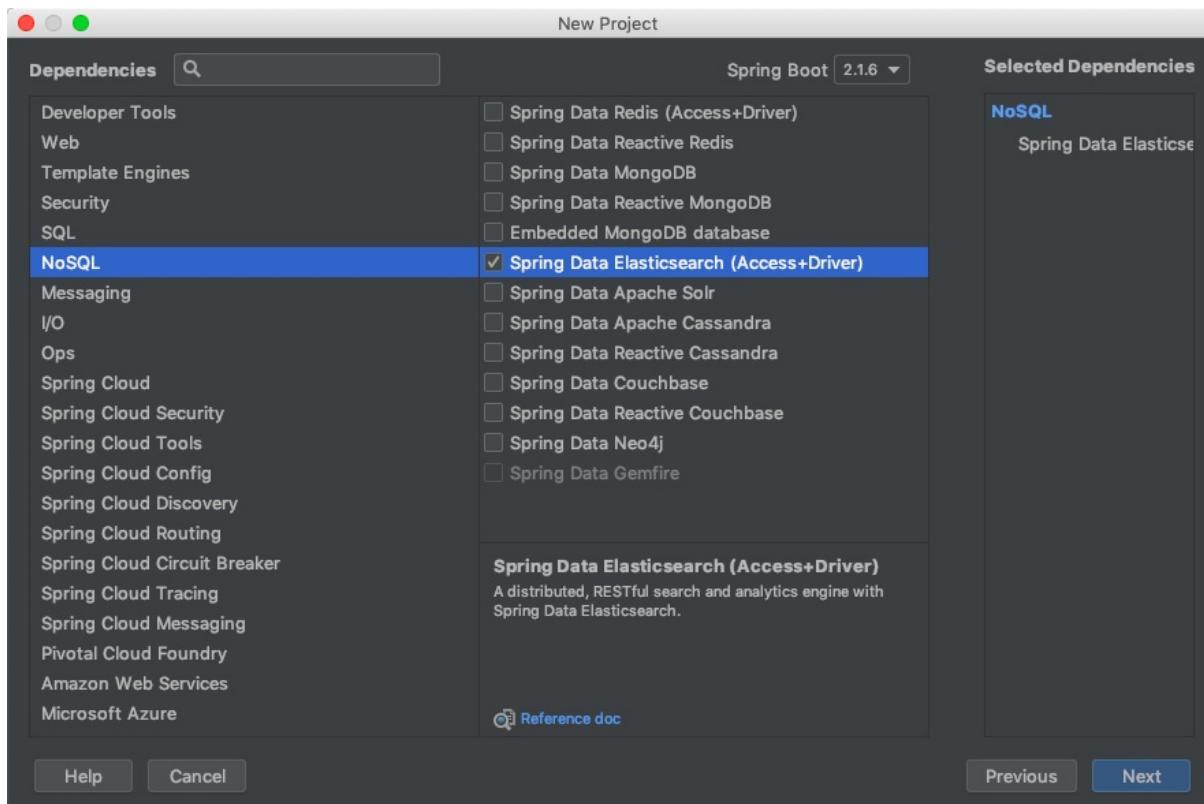
作为简单的API和简单搜索两章节的启下部分，本节示例仍然是基于上一章节的示例。

通过IDEA创建Spring Boot工程，并且在创建过程中选择 Spring Data Elasticsearch ，主要步骤如下图所示：

第一步，创建工程，选择 Spring Initializr 。



第二步，选择SpringBoot的依赖 NoSQL -> Spring Data ElasticSearch。



创建好Spring Data ElasticSearch的Spring Boot工程后，按照ES惯例是定义Index以及Type和Mapping。在 Spring Data ElasticSearch 中定义Index、Type以及Mapping非常简单。ES文档数据实质上对应的是一个数据结构，也就是在 Spring Data ElasticSearch 要我们把ES中的文档数据模

型与Java对象映射关联。

定义StudentPO对象，对象中定义Index以及Type， Mapping映射我们引入外部json文件（json格式的Mapping就是在简单搜索一章中定义的Mapping数据）。

```
package com.coderbuff.es.easy.domain;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.elasticsearch.annotations.Document;
import org.springframework.data.elasticsearch.annotations.Field;
import org.springframework.data.elasticsearch.annotations.FieldType;
import org.springframework.data.elasticsearch.annotations.Mapping;

import java.io.Serializable;

/**
 * ES mapping映射对应的PO
 * Created by OKevin on 2019-06-26 22:52
 */
@Getter
@Setter
@ToString
@Document(indexName = "user", type = "student")
@Mapping(mappingPath = "student_mapping.json")
public class StudentPO implements Serializable {

 private String id;

 /**
 * 姓名
 */
 private String name;

 /**
 * 年龄
 */
 private Integer age;
}
```

Spring Data Elasticsearch 为我们屏蔽了操作ES太多的细节，以至于真的就是开箱即用，它操作ES主要是通过 ElasticsearchRepository 接口，我们在定义自己具体业务时，只需要继承它，扩展自己的方法。

```
package com.coderbuff.es.easy.dao;

import com.coderbuff.es.easy.domain.StudentPO;
```

```

import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;
import org.springframework.stereotype.Repository;

/**
 * Created by OKevin on 2019-06-26 23:45
 */
@Repository
public interface StudentRepository extends ElasticsearchRepository<StudentPO, String> {
}

```

ElasticsearchTemplate 可以说是 Spring Data ElasticSearch 最为重要的一个类，它对ES的 Java API进行了封装，创建索引等都离不开它。在Spring中要使用它，必然是要先注入，也就是实例化一个bean。而 Spring Data ElasticSearch 早为我们做好了一切，只需要在 application.properties 中定义 spring.data.elasticsearch.cluster-nodes=127.0.0.1:9300 ，就可大功告成（网上有人的教程还在使用applicationContext.xml定义一个 bean，事实证明，受到了Spring多年的“毒害”，Spring Boot远比我们想象的智能）。

单元测试创建Index、Type以及定义Mapping。

```

package com.coderbuff.es;

import com.coderbuff.es.easy.domain.StudentPO;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.elasticsearch.core.ElasticsearchTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringDataElasticsearchApplicationTests {

 @Autowired
 private ElasticsearchTemplate elasticsearchTemplate;

 /**
 * 测试创建Index, type和Mapping定义
 */
 @Test
 public void createIndex() {
 elasticsearchTemplate.createIndex(StudentPO.class);
 elasticsearchTemplate.putMapping(StudentPO.class);
 }
}

```

使用 `GET http://localhost:9200/user` 请求命令，可看到通过 Spring Data ElasticSearch 创建的索引。

索引创建完成后，接下来就是定义操作student文档数据的接口。在 `StudentService` 接口的实现中，通过组合 `StudentRepository` 类对ES进行操作。`StudentRepository` 类继承了 `ElasticsearchRepository` 接口，这个接口的实现已经为我们提供了基本的数据操作，保存、修改、删除只是一句代码的事。就算查询、分页也为我们提供好了builder类。“最难”的实际上不是实现这些方法，而是如何构造查询参数 `SearchQuery`。创建 `SearchQuery` 实例，有两种方式：

1. 构建 `NativeSearchQueryBuilder` 类，通过链式调用构造查询参数。
2. 构建 `NativeSearchQuery` 类，通过构造方法传入查询参数。

这里以“不分页range范围和term查询age>=21且age<26且name=kevin”为例。

```
SearchQuery searchQuery = new NativeSearchQueryBuilder()
 .withQuery(QueryBuilders.boolQuery()
 .must(QueryBuilders.rangeQuery("age").gte(21).lt(26))
 .must(QueryBuilders.termQuery("name", "kevin")))).build();
```

搜索条件的构造一定要对ES的查询结构有比较清晰的认识，如果是在了解了简单的API和简单搜索两章的前提下，学习如何构造多加练习一定能掌握。这里就不一一验证前面章节的示例，一定要配合代码使用练习(<https://github.com/yu-linfeng/elasticsearch6.xTutorial/tree/master/code/spring-data-elasticsearch>)

## TransportClient

ES的Java API非常广泛，一种操作可能会有好几种写法。Spring Data ElasticSearch实际上是对ES Java API的再次封装，从使用上将更加简单。

本节请直接对照代码学习使用，如果要讲解ES的Java API那将是一个十分庞大的工作，<https://github.com/yu-linfeng/elasticsearch6.xTutorial/tree/master/code/transportclient-elasticsearch>

# 父-子关系文档

打虎亲兄弟，上阵父子兵。

本章作为复杂搜索的铺垫，介绍父子文档是为了更好的介绍复杂场景下的ES操作。

在非关系型数据库数据库中，我们常常会有表与表的关联查询。例如学生表和成绩表的关联查询就能查出学会的信息和成绩信息。在ES中，父子关系文档就类似于表的关联查询。

## 背景

ES5.x开始借助父子关系文档实现多表关联查询，核心是一个索引Index下可以创建多个类型Type。但ES6.x开始只允许一个索引Index下创建一个类型Type，甚至在未来的版本中将会移除创建类型Type。为了继续支持多表关联查询，ES6.x推出了join新类型来支持父子关系文档的创建。

## 问题

假设现在有这样的需求场景：一个博客有多篇文章，文章有标题、内容、作者、日期等信息，同时一篇文章中会有评论，评论有评论的内容、作者、日期等信息，通过ES来存储博客的文章及评论信息。

此时文章本身就是“父”，而评论就是“子”，这类问题也可以通过nested嵌套对象实现，大部分情况下nested嵌套对象和parent-child父子对象能够互相替代，但他们仍然不同的优缺点。下面将介绍这两种数据结构。

## nested嵌套对象

一篇文章的数据结构如下图所示：

```
{
 "title": "ElasticSearch6.x实战教程",
 "author": "OKevin",
 "content": "这是一篇水文",
 "created": 1562141626000,
 "comments": [
 {
 "name": "张三",
 "content": "写的真菜",
 "created": 1562141689000
 },
 {
 "name": "李四",
 "content": "辣鸡",
 "created": 1562141745000
 }
]
}
```

通过RESTful API创建索引及定义映射结构：

```
PUT http://localhost:9200/blog
{
 "mappings": {
 "article": {
 "properties": {
 "title": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 },
 "author": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 },
 "content": {
 "type": "text",
 "analyzer": "ik_smart"
 },
 "created": {
 "type": "date"
 },
 "comments": {
 "type": "nested",
 "properties": {
 "name": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 }
 }
 },
 "content": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {

```

```
 "type": "keyword",
 "ignore_above": 256
 }
}
},
"created": {
 "type": "date"
}
}
}
}
}
}
```

插入数据:

```
POST http://localhost:9200/blog/article
{
 "title": "ElasticSearch6.x实战教程",
 "author": "OKevin",
 "content": "这是一篇水文",
 "created": 1562141626000,
 "comments": [
 {
 "name": "张三",
 "content": "写的真菜",
 "created": 1562141689000
 },
 {
 "name": "李四",
 "content": "辣鸡",
 "created": 1562141745000
 }
]
}
```

```
POST http://localhost:9200/blog/article
{
 "title": "ElasticSearch6.x从入门到放弃",
 "author": "OKevin",
 "content": "这是一篇ES从入门到放弃文章",
 "created": 1562144089000,
 "comments": [
 {
 "name": "张三",
 "content": "我已入门",
 "created": 1562144089000
 },
 {
 "name": "李四",
 "content": "我已放弃",
 "created": 1562144089000
 }
]
}
```

```
}
```

```
POST http://localhost:9200/blog/article
{
 "title": "ElasticSearch6.x原理解析",
 "author": "专家",
 "content": "这是一篇ES原理解析的文章",
 "created": 1562144089000,
 "comments": [
 {
 "name": "张三",
 "content": "牛逼，专家就是不一样",
 "created": 1562144089000
 },
 {
 "name": "李四",
 "content": "大牛",
 "created": 1562144089000
 }
]
}
```

### 1. 查询作者为“OKevin”文章的所有评论（父查子）

```
GET http://localhost:9200/blog/article/_search
{
 "query": {
 "bool": {
 "must": [
 {
 "match": {
 "author.keyword": "OKevin"
 }
 }
]
 }
 }
}
```

ES结果返回2条作者为“OKevin”的全部数据。

### 1. 查询评论中含有“辣鸡”的文章（子查父）

```
GET http://localhost:9200/blog/article/_search
{
 "query": {
 "bool": {
 "must": [
 {
 "match": {
 "author.keyword": "OKevin"
 }
 }
],
 "nested": {
 "path": "comments"
 }
 }
 }
}
```

```

 "path": "comments",
 "query": {
 "bool": {
 "must": [
 {
 "match": {
 "comments.content": "辣鸡"
 }
 }
]
 }
 }
}

```

ES确实只返回了包含"辣鸡"的数据。

两次查询都直接返回了整个文档数据。

## parent-child父子文档

既然父子文档能实现表的关联查询，那它的数据结构就应该是这样：

文章数据结构

```
{
 "title": "ElasticSearch6.x实战教程",
 "author": "OKevin",
 "content": "这是一篇实战教程",
 "created": 1562141626000,
 "comments": []
}
```

评论数据结构

```
{
 "name": "张三",
 "content": "写的真菜",
 "created": 1562141689000
}
```

ES6.x以前是将这两个结构分别存储在两个类型Type中关联(这看起来更接近关系型数据库表与表的关联查询)，但在ES6.x开始一个索引Index只能创建一个类型Type，要再想实现表关联查询，就意味着需要把上述两张表揉在一起，ES6.x由此定义了一个新的数据类型——join。

通过RESTful API创建索引及定义映射结构：

```
{
 "mappings": {
 "article": {
 "properties": {
 "title": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 },
 "author": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 },
 "content": {
 "type": "text",
 "analyzer": "ik_smart"
 },
 "created": {
 "type": "date"
 },
 "comments": {
 "type": "join",
 "relations": {
 "article": "comment"
 }
 }
 }
 }
 }
}
```

重点关注其中的"comments"字段，可以看到类型定义为 `join`，`relations`定义了谁是父谁是子，"article":"comment"表示article是父comment是子。

父子文档的插入是父与子分别插入(因为可以理解为把多个表塞到了一张表里)。

插入父文档：

```
POST http://localhost:9200/blog/article/1
```

```
{
 "title": "ElasticSearch6.x实战教程",
 "author": "OKevin",
 "content": "这是一篇水文",
 "created": 1562141626000,
 "comments": "article"
}
```

```
POST http://localhost:9200/blog/article/2
{
 "title": "ElasticSearch6.x从入门到放弃",
 "author": "OKevin",
 "content": "这是一篇ES从入门到放弃文章",
 "created": 1562144089000,
 "comments": "article"
}
```

```
POST http://localhost:9200/blog/article/3
{
 "title": "ElasticSearch6.x原理解析",
 "author": "专家",
 "content": "这是一篇ES原理解析的文章",
 "created": 1562144089000,
 "comments": "article"
}
```

插入子文档：

```
POST http://localhost:9200/blog/article/4?routing=1
{
 "name": "张三",
 "content": "写的真菜",
 "created": 1562141689000,
 "comments": {
 "name": "comment",
 "parent": 1
 }
}
```

```
POST http://localhost:9200/blog/article/5?routing=1
{
 "name": "李四",
 "content": "辣鸡",
 "created": 1562141745000,
 "comments": {
 "name": "comment",
 "parent": 1
 }
```

```
 }
}
```

```
POST http://localhost:9200/blog/article/6?routing=2
{
 "name": "张三",
 "content": "我已入门",
 "created": 1562144089000,
 "comments": {
 "name": "comment",
 "parent": 2
 }
}
```

```
POST http://localhost:9200/blog/article/7?routing=2
{
 "name": "李四",
 "content": "我已放弃",
 "created": 1562144089000,
 "comments": {
 "name": "comment",
 "parent": 2
 }
}
```

```
POST http://localhost:9200/blog/article/8?routing=3
{
 "name": "张三",
 "content": "牛逼，专家就是不一样",
 "created": 1562144089000,
 "comments": {
 "name": "comment",
 "parent": 3
 }
}
```

```
POST http://localhost:9200/blog/article/9?routing=3
{
 "name": "李四",
 "content": "大牛",
 "created": 1562144089000,
 "comments": {
 "name": "comment",
 "parent": 3
 }
}
```

如果查询索引数据会发现一共有9条数据，并不是 nested 那样将“评论”嵌套“文章”中的。

### 1. 查询作者为“OKevin”文章的所有评论（父查子）

```
GET http://localhost:9200/blog/article/_search
{
 "query": {
 "has_parent": {
 "parent_type": "article",
 "query": {
 "match": {
 "author.keyword": "OKevin"
 }
 }
 }
 }
}
```

ES只返回了comment评论结构中的数据，而不是全部包括文章数据也返回。这是嵌套对象查询与父子文档查询的区别之——子文档可以单独返回。

### 1. 查询评论中含有“辣鸡”的文章（子查父）

```
GET http://localhost:9200/blog/article/_search
{
 "query": {
 "has_child": {
 "type": "comment",
 "query": {
 "match": {
 "content": "辣鸡"
 }
 }
 }
 }
}
```

ES同样也只返回了父文档的数据，而没有子文档(评论)的数据。

nested 嵌套对象和 parent-child 父子文档之间最大的区别，嵌套对象中的“父子”是一个文档数据，而父子文档的中的“父子”是两个文档数据。这意味着嵌套对象中如果涉及对嵌套文档的操作会对整个文档造成影响（重新索引，但查询快），包括修改、删除、查询。而父子文档子文档或者父文档本身就是独立的文档，对子文档或者父文档的操作并不会相互影响（不会重新索引，查询相对慢）。

# 复杂搜索

黑夜给了我黑色的眼睛，我却用它寻找光明。

经过了解简单的API和简单搜索，已经基本上能应付大部分的使用场景。可是非关系型数据库数据的文档数据往往又多又杂，各种各样冗余的字段，组成了一条“记录”。复杂的数据结构，带来的就是复杂的搜索。所以在进入本章节前，我们要构建一个尽可能“复杂”的数据结构。

下面分为两个场景，场景1偏向数据结构上的复杂并且介绍聚合查询、指定字段返回、深分页，场景2偏向搜索精度上的复杂。

## 场景1

存储一个公司的员工，员工信息包含姓名、工号、性别、出生年月日、岗位、上级、下级、所在部门、进入公司时间、修改时间、创建时间。其中员工工号作为主键ID全局唯一，员工只有一个直属上级，但有多个下级，可以通过父子文档实现。员工有可能属于多个部门（特别是领导可能兼任多个部门的负责人）。

### 数据结构

创建索引并定义映射结构：

```
PUT http://localhost:9200/company
{
 "mappings": {
 "employee": {
 "properties": {
 "id": {
 "type": "keyword"
 },
 "name": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 },
 "sex": {
 "type": "keyword"
 },
 "age": {
 "type": "integer"
 },
 "birthday": {
 "type": "date"
 }
 }
 }
 }
}
```

```
 "type": "date"
 },
 "position": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 },
 "level": {
 "type": "join",
 "relations": {
 "superior": "staff",
 "staff": "junior"
 }
 },
 "departments": {
 "type": "text",
 "analyzer": "ik_smart",
 "fields": {
 "keyword": {
 "type": "keyword",
 "ignore_above": 256
 }
 }
 },
 "joinTime": {
 "type": "date"
 },
 "modified": {
 "type": "date"
 },
 "created": {
 "type": "date"
 }
}
}
```

## 数据

接下来是构造数据，我们构造几条关键数据。

- 张三是公司的董事长，他是最大的领导，不属于任何部门。
- 李四的上级是张三，他的下级是王五、赵六、孙七、周八，他同时是市场部和研发部的负责人，

也就是隶属于市场部和研发部。

- 王五、赵六的上级是张三，他没有下级，他隶属于市场部。
- 孙七、周八的上级是李四，他没有下级，他隶属于研发部。

更为全面直观的数据如下表所示：

姓名	工号	性别	年龄	出生年月日	岗位	上级	下级	部门	进入公司时间	修改时间
张三	1	男	49	1970-01-01	董事长	/	李四	/	1990-01-01	1562167817000
李四	2	男	39	1980-04-03	总经理	张三	王五、赵六、孙七、周八	市场部、研发部	2001-02-02	1562167817000
王五	3	女	27	1992-09-01	销售	李四	/	市场部	2010-07-01	1562167817000
赵六	4	男	29	1990-10-10	销售	李四	/	市场部	2010-08-08	1562167817000
孙七	5	男	26	1993-12-10	前端工程师	李四	/	研发部	2016-07-01	1562167817000
周八	6	男	25	1994-05-11	Java工程师	李四	/	研发部	2018-03-10	1562167817000

插入6条数据：

```
POST http://localhost:9200/company/employee/1?routing=1
{
 "id": "1",
 "name": "张三",
 "sex": "男",
 "age": 49,
 "birthday": "1970-01-01",
 "position": "董事长",
 "level": {
 "name": "superior"
 },
 "joinTime": "1990-01-01",
 "modified": "1562167817000",
 "created": "1562167817000"
}
```

```
POST http://localhost:9200/company/employee/2?routing=1
{
```

```
"id":"2",
"name":"李四",
"sex":"男",
"age":39,
"birthday":"1980-04-03",
"position":"总经理",
"level":{
"name":"staff",
"parent":"1"
},
"departments":["市场部", "研发部"],
"joinTime":"2001-02-02",
"modified":"1562167817000",
"created":"1562167817000"
}
```

```
POST http://localhost:9200/company/employee/3?routing=1
{
 "id":"3",
 "name":"王五",
 "sex":"女",
 "age":27,
 "birthday":"1992-09-01",
 "position":"销售",
 "level":{
 "name":"junior",
 "parent":"2"
 },
 "departments":["市场部"],
 "joinTime":"2010-07-01",
 "modified":"1562167817000",
 "created":"1562167817000"
}
```

```
POST http://localhost:9200/company/employee/4?routing=1
{
 "id":"4",
 "name":"赵六",
 "sex":"男",
 "age":29,
 "birthday":"1990-10-10",
 "position":"销售",
 "level":{
 "name":"junior",
 "parent":"2"
 },
 "departments":["市场部"],
 "joinTime":"2010-08-08",
 "modified":"1562167817000",
 "created":"1562167817000"
}
```

```

 "modified":"1562167817000",
 "created":"1562167817000"
}
```

```

POST http://localhost:9200/company/employee/5?routing=1
{
 "id":"5",
 "name":"孙七",
 "sex":"男",
 "age":26,
 "birthday":"1993-12-10",
 "position":"前端工程师",
 "level":{
 "name":"junior",
 "parent":"2"
 },
 "departments":["研发部"],
 "joinTime":"2016-07-01",
 "modified":"1562167817000",
 "created":"1562167817000"
}
```

```

POST http://localhost:9200/company/employee/6?routing=1
{
 "id":"6",
 "name":"周八",
 "sex":"男",
 "age":28,
 "birthday":"1994-05-11",
 "position":"Java工程师",
 "level":{
 "name":"junior",
 "parent":"2"
 },
 "departments":["研发部"],
 "joinTime":"2018-03-10",
 "modified":"1562167817000",
 "created":"1562167817000"
}
```

## 搜索

### 1. 查询研发部的员工

```

GET http://localhost:9200/company/employee/_search
{
 "query":{
```

```

 "match": {
 "departments": "研发部"
 }
}
}

```

### 1. 查询在研发部且在市场部的员工

```

GET http://localhost:9200/company/employee/_search
{
 "query": {
 "bool": {
 "must": [
 {
 "match": {
 "departments": "市场部"
 }
 },
 {
 "match": {
 "departments": "研发部"
 }
 }
]
 }
 }
}

```

\*被搜索的字段是一个数组类型，但对查询语句并没有特殊的要求。

### 1. 查询name="张三"的直接下属。

```

GET http://localhost:9200/company/employee/_search
{
 "query": {
 "has_parent": {
 "parent_type": "superior",
 "query": {
 "match": {
 "name": "张三"
 }
 }
 }
 }
}

```

### 1. 查询name="李四"的直接下属。

```

GET http://localhost:9200/company/employee/_search
{
 "query": {

```

```

"has_parent": {
 "parent_type": "staff",
 "query": {
 "match": {
 "name": "李四"
 }
 }
}
}
}

```

1. 查询name="王五"的直接上级。

```

GET http://localhost:9200/company/employee/_search
{
 "query": {
 "has_child": {
 "type": "junior",
 "query": {
 "match": {
 "name": "王五"
 }
 }
 }
 }
}

```

## 聚合查询

ES中的聚合查询类似MySQL中的聚合函数(avg、max等)，例如计算员工的平均年龄。

```

GET http://localhost:9200/company/employee/_search?pretty
{
 "size": 0,
 "aggs": {
 "avg_age": {
 "avg": {
 "field": "age"
 }
 }
 }
}

```

## 指定字段查询

指定字段返回值在查询结果中指定需要返回的字段。例如只查询张三的生日。

```

GET http://localhost:9200/company/employee/_search?pretty

```

```
{
 "_source": ["name", "birthday"],
 "query": {
 "match": {
 "name": "张三"
 }
 }
}
```

## 深分页

ES的深分页是一个老生常谈的问题。用过ES的都知道，ES默认查询深度不能超过10000条，也就是`page * pageSize < 10000`。如果需要查询超过1万条的数据，要么通过设置最大深度，要么通过`scroll`滚动查询。如果调整配置，即使能查出来，性能也会很差。但通过`scroll`滚动查询的方式带来的问题就是只能进行"上一页"、"下一页"的操作，而不能进行页码跳转。

首先需要初始化查询

```
```json
GET http://localhost:9200/company/employee/_search?scroll=1m
{
  "query": {
    "match_all": {}
  },
  "size": 1,
  "_source": ["id"]
}
```

像普通查询结果一样进行查询，url中的`scroll=1m`指的是游标查询的过期时间为1分钟，每次查询就会更新，设置过长会用过多的时间。

接下来就可以通过上述API返回的`_scroll_id`进行滚动查询，假设上面的结果返回`"_scroll_id": "DnF1ZXJ5VGh1bkZldGNoBQAAAAAAAABFK1pNzdFUVhDU3hxX3VtSVFUDJBW1EAAAAAAABQhZNaTc3RVFYQ1N4cV91bU1RVHQyQVpR"`。

```
GET http://localhost:9200/_search/scroll
{
  "scroll": "1m",
  "scroll_id": "DnF1ZXJ5VGh1bkZldGNoBQAAAAAAAABFK1pNzdFUVhDU3hxX3VtSVFUDJBW1EAA
  AAAAABQhZNaTc3RVFYQ1N4cV91bU1RVHQyQVpRAAAAAAAUMWTwk3N0VRWENTeHffdw1JUVR0MkFaUQAA
  AAAAABRRZNaTc3RVFYQ1N4cV91bU1RVHQyQVpR"
}
```

这种方式有一个小小的弊端，如果超过过期时间就不能继续往下查询，这种查询适合一次全量查询所有数据。但现实情况有可能是用户在一个页面停留很长时间，再点击上一页或者下一页，此时超过过期时间页面不能再进行查询。所以还有另外一种方式，范围查询。

另一种深分页

假设员工数据中的工号ID是按递增且唯一的顺序，那么我们可以通过范围查询进行分页。

例如，按ID递增排序，第一查询ID>0的数据，数据量为1。

```
GET http://localhost:9200/company/employee/_search
{
  "query": {
    "range": {
      "id": {
        "gt": 0
      }
    },
    "size": 1,
    "sort": {
      "id": {
        "order": "asc"
      }
    }
  }
}
```

此时返回ID=1的1条数据，我们再继续查询ID>1的数据，数据量仍然是1。

```
GET http://localhost:9200/company/employee/_search
{
  "query": {
    "range": {
      "id": {
        "gt": 1
      }
    },
    "size": 1,
    "sort": {
      "id": {
        "order": "asc"
      }
    }
  }
}
```

这样我们同样做到了深分页的查询，并且没有过期时间的限制。

场景2

存储商品数据，根据商品名称搜索商品，要求准确度高，不能搜索洗面奶结果出现面粉。

由于这个场景主要涉及的是搜索的精度问题，所以并不会有复杂的数据结构，只有一个title字段。

定义一个只包含title字段且分词器默认为 standard 的索引：

```
PUT http://localhost:9200/ware_index
{
  "mappings": {
    "ware": {
      "properties": {
        "title": {
          "type": "text"
        }
      }
    }
  }
}
```

插入两条数据：

```
POST http://localhost:9200/ware_index/ware
{
  "title": "洗面奶"
}
```

```
POST http://localhost:9200/ware_index/ware
{
  "title": "面粉"
}
```

搜索关键字"洗面奶"：

```
POST http://localhost:9200/ware_index/ware/_search
{
  "query": {
    "match": {
      "title": "洗面奶"
    }
  }
}
```

搜索结果出现了"洗面奶"和"面粉"两个风马牛不相及的结果，这显然不符合我们的预期。

原因在分词一章中已经说明，`text` 类型默认分词器为 `standard`，它会将中文字符串一个字一个字拆分，也就是将“洗面奶”拆分成了“洗”、“面”、“奶”，将“面粉”拆分成了“面”、“粉”。而 `match` 会将搜索的关键词拆分，也就拆分成了“洗”、“面”、“奶”，最后两个“面”都能匹配上，也就出现了上述结果。所以对于中文的字符串搜索我们需要指定分词器，而常用的分词器是 `ik_smart`，它会按照最大粒度拆分，如果采用 `ik_max_word` 它会将词按照最小粒度拆分，也有可能造成上述结果。

```
DELETE http://localhost:9200/ware_index 删除索引，重新创建并指定title字段的分词器
为 ik_smart 。
```

```
PUT http://localhost:9200/ware_index
{
  "mappings": {
    "ware": {
      "properties": {
        "id": {
          "type": "keyword"
        },
        "title": {
          "type": "text",
          "analyzer": "ik_smart"
        }
      }
    }
  }
}
```

这时如果插入“洗面奶”和“面粉”，搜索“洗面奶”是结果就只有一条。但此时我们插入以下两条数据：

```
POST http://localhost:9200/ware_index/ware
{
  "id": "1",
  "title": "新希望牛奶"
}
```

```
POST http://localhost:9200/ware_index/ware
{
  "id": "2",
  "title": "春秋上新短袖"
}
```

搜索关键字“新希望牛奶”：

```
POST http://localhost:9200/ware_index/ware/_search
{
  "query": {
    "match": {
      "title": "新希望牛奶"
    }
  }
}
```

```

        }
    }
}
```

搜索结果出现了刚插入的2条，显然第二条“春秋上新短袖”并不是我们想要的结果。出现这种问题的原因同样是因为分词的问题，在 `ik` 插件的词库中并没有“新希望”一词，所以它会把搜索的关键词“新希望”拆分为“新”和“希望”，同样在“春秋上新短袖”中“新”也并没有组合成其它词语，它也被单独拆成了“新”，这就造成了上述结果。解决这个问题的办法当然可以在 `ik` 插件中新增“新希望”词语，如果我们在分词中所做的那样，但也有其它的办法。

短语查询

`match_phrase`，短语查询，它会将搜索关键字“新希望牛奶”拆分成一个词项列表“新 希望 牛奶”，对于搜索的结果需要完全匹配这些词项，且位置对应，本例中的“新希望牛奶”文档数据从词项和位置上完全对应，故通过 `match_phrase` 短语查询可搜索出结果，且只有一条数据。

```

POST http://localhost:9200/ware_index/ware/_search
{
  "query": {
    "match_phrase": {
      "title": "新希望牛奶"
    }
  }
}
```

尽管这能满足我们的搜索结果，但是用户实际在搜索中常常可能是“牛奶 新希望”这样的顺序，但遗憾的是根据 `match_phrase` 短语匹配的要求是需要被搜索的文档需要完全匹配词项且位置对应，关键字“牛奶 新希望”被解析成了“牛奶 新 希望”，尽管它与“新希望牛奶”词项匹配但位置没有对应，所以并不能搜索出任何结果。同理，此时如果我们插入“新希望的牛奶”数据时，无论是搜索“新希望牛奶”还是“牛奶新希望”均不能搜索出“新希望的牛奶”结果，前者的关键字是因为词项没有完全匹配，后者的关键字是因为词项和位置没有完全匹配。

所以 `match_phrase` 也没有达到完美的效果。

短语前缀查询

`match_phrase_prefix`，短语前缀查询，类似MySQL中的 `like "新希望%"`，它大体上和 `match_phrase_prefix` 一致，也是需要满足文档数据和搜索关键字在词项和位置上保持一致，同样如果搜索“牛奶新希望”也不会出现任何结果。它也并没有达到我们想要的结果。

最低匹配度

前面两种查询中虽然能通过“新希望牛奶”搜索到我们想要的结果，但是对于“牛奶 新希望”却无能为力。接下来的这种查询方式能“完美”的达到我们想要的效果。

先来看最低匹配度的查询示例：

```
POST http://localhost:9200/ware_index/ware/_search
{
  "query": {
    "match": {
      "title": {
        "query": "新希望牛奶",
        "minimum_should_match": "80%"
      }
    }
  }
}
```

`minimum_should_match` 即最低匹配度。"80%"代表什么意思呢？还是要从关键字"新希望牛奶"被解析成哪几个词项说起，前面说到"新希望牛奶"被解析成"新 希望 牛奶"三个词项，如果通过 `match` 搜索，则含有"新"的数据同样出现在搜索结果中。"80%"的含义则是3个词项必须至少匹配 $80\% \cdot 3 = 2.4$ 个词项才会出现在搜索结果中，向下取整为2，即搜索的数据中需要至少包含2个词项。显然，"春秋上新短袖"只有1个词项，不满足*最低匹配度2个词项的要求，故不会出现在搜索结果中。

同样，如果搜索"牛奶 新希望"也是上述的结果，它并不是短语匹配，所以并不会要求词项所匹配的位置相同。

可以推出，如果 `"minimum_should_match": "100%"` 也就是要求完全匹配，此时要求数据中包含所有的词项，这样会出现较少的搜索结果；如果 `"minimum_should_match: 0"` 此时并不代表一个词项都可以不包含，而是只需要有一个词项就能出现在搜索结果，实际上就是默认的 `match` 搜索，这样会出现较多的搜索结果。

找到一个合适的值，就能有一个较好的体验，根据二八原则，以及实践表明，设置为"80%"能满足大部分场景，既不会多出无用的搜索结果，也不会少。

Java客户端（下）

基于[Java客户端（上）](#)，本文不再赘述如何创建一个Spring Data ElasticSearch工程，也不再做过多文字叙述。更多的请一定配合源码使用，源码地址<https://github.com/yulinfeng/elasticsearch6.xTutorial/tree/master/code/spring-data-elasticsearch>），具体代码目录在 complex 包。

本章请一定结合代码重点关注如何通过Java API进行父子文档的数据插入，以及查询。

父子文档的数据插入

父子文档在ES中存储的格式实际上是以键值对方式存在，例如在定义映射Mapping时，我们将子文档定义为：

```
{
    .....
    "level": {
        "type": "join",
        "relations": {
            "superior": "staff",
            "staff": "junior"
        }
    }
    .....
}
```

在写入一条数据时：

```
{
    .....
    "level": {
        "name": "staff",
        "parent": "1"
    }
    .....
}
```

对于Java实体，我们可以把 level 字段设置为 Map<String, Object> 类型。关键注意的是，在使用Spring Data ElasticSearch时，我们不能直接调用 save 或者 saveAll 方法。ES规定父子文档必须属于同一分片，也就是说在写入子文档时，需要定义 routing 参数。下面是代码节选：

```
BulkRequestBuilder bulkRequestBuilder = client.prepareBulk();
bulkRequestBuilder.add(client.prepareIndex("company", "employee", employeeP0.getId())
    .setRouting(routing).setSource(mapper.writeValueAsString(employeeP0), XContentType.JSON))
    .execute().actionGet();
```

一定参考源码一起使用。

ES实在是一个非常强大的搜索引擎。能力有限，实在不能将所有的Java API一一举例讲解，如果你在编写代码时，遇到困难也请联系作者邮箱**hellobug at outlook.com**，或者通过公众号**coderbuff**，解答得了的一定解答，解答不了的一起解答。

实战：ELK日志分析系统

ElasticSearch、Logstash、Kibana简称ELK系统，主要用于日志的收集与分析。

一个完整的大型分布式系统，会有很多与业务不相关的系统，其中日志系统是不可或缺的一个，集中式日志系统需要收集来自不同服务的日志，对它进行集中管理存储以及分析。ELK就是这样一个系统。

ElasticSearch是一个开源分布式搜索引擎，在ELK系统中提供对数据的搜索、分析、存储。

Logstash主要用于日志的收集，在ELK系统中作为日志数据源的传输。

Kibana则是一个可视化管理工具，在ELK系统中起可视化分析查看的作用。

安装部署ELK

ElasticSearch

ElasticSearch的安装在“准备工作”中已经说明，这里不再赘述。

Kibana

Kibana6.3.2下载地址（Linux、mac OS、Windows对应不同的版本）：<https://www.elastic.co/cn/downloads/past-releases/kibana-6-3-2>。Logstash历史版本下载页面：<https://www.elastic.co/cn/downloads/past-releases#kibana>。

mac OS

1. 通过命令 `tar -zxvf kibana-6.3.2-darwin-x86_64.tar.gz` 解压到当前用户目录(或者其它位置)。
2. 解压后进入 `kibana-6.3.2-darwin-x86_64` 目录，执行 `vim config/kibana.yml` 命令，修改配置（注意 `yml` 格式的配置文件冒号必须有英文空格）。

```
server.port: 5601
server.host: "localhost"
elasticsearch.url: "http://localhost:9200"
logging.dest: /Users/yulinfeng/log/kibana.log
```

1. 执行 `./bin/kibana` 命令启动Kibana。

启动完成，打开浏览器输入 `localhost:5061`。

The screenshot shows the Kibana interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timeline, APM, Dev Tools, Monitoring, and Management. Below the sidebar is a 'Collapse' button.

Add Data to Kibana

This section contains four cards:

- APM**: APM automatically collects in-depth performance metrics and errors from inside your applications. Includes a 'Add APM' button.
- Logging**: Ingest logs from popular data sources and easily visualize in preconfigured dashboards. Includes a 'Add log data' button.
- Metrics**: Collect metrics from the operating system and services running on your servers. Includes a 'Add metric data' button.
- Security Analytics**: Centralize security events for interactive investigation in ready-to-go visualizations. Includes a 'Add security events' button.

Data already in Elasticsearch? [Set up index patterns](#)

Visualize and Explore Data

This section contains four cards:

- APM**: Automatically collect in-depth performance metrics and errors from inside your applications.
- Discover**: Interactively explore your data by querying and filtering raw documents.
- Machine Learning**: Automatically model the
- Dashboard**: Display and share a collection of visualizations and saved searches.
- Graph**: Surface and analyze relevant relationships in your Elasticsearch data.
- Timeline**: Use an expression language

Manage and Administer the Elastic Stack

This section contains six cards:

- Console**: Skip cURL and use this JSON interface to work with your data directly.
- Monitoring**: Track the real-time health and performance of your Elastic Stack.
- Index Patterns**: Manage the index patterns that help retrieve your data from Elasticsearch.
- Saved Objects**: Import, export, and manage your saved searches, visualizations, and dashboards.
- Security Settings**: Protect your data and easily
- Watcher**: Detect changes in your data

Linux

安装过程同mac OS。

Logstash

Logstash6.3.2下载地址（Linux、mac OS、Windows通用，下载zip包即可）：<https://www.elastic.co/cn/downloads/past-releases/logstash-6-3-2>。Logstash历史版本下载页面：<https://www.elastic.co/cn/downloads/past-releases#logstash>。

macOS

1. 解压 `logstash-6.3.2.zip` 解压到当前用户目录(或者其它位置)
2. 解压后进入 `logstash-6.3.2` 目录，执行 `vim logstash.conf` 配置文件。

```
input {
  tcp {
    mode => "server"
    host => "127.0.0.1"
    port => 4568
  }
}
output {
  elasticsearch {
    hosts => ["localhost:9200"]
    index => "log"
    document_type => "log4j2"
  }
}
```

这个配置的含义为，Logstash的日志采集来源是 4568 端口（后面我们会通过程序代码通过log4j向端口 4568 打印日志）。Logstash的日志向ElasticSearch输出。

1. 执行 `./bin/logstash -f logstash.conf` 命令启动Logstash。

Linux

安装过程同mac OS。

日志源DEMO

在部署Logstash时，我们定义了日志的来源是端口 4568，接下来我们模拟一个程序利用log4j2通过socket连接将日志发送到 4568 端口，DEMO完整代码地址：<https://github.com/yulinfeng/elasticsearch6.xTutorial/tree/master/code/logstash>。

DEMO的逻辑只有打印日志，主要在 `log4j2.xml` 需要配置socket方式打印。

```
.....
<Socket name="logstash-tcp" host="localhost" port="4568" protocol="TCP">
  <PatternLayout pattern="${LOG_PATTERN}" />
</Socket>
.....
```

详细代码直接查看源码<https://github.com/yulinfeng/elasticsearch6.xTutorial/tree/master/code/logstash>。

启动Spring Boot程序后，控制台开始输出日志，此时返回浏览器查看 `localhost:5061`，点击 Management 菜单，Index pattern 中输入"log"(即定义的索引Index)，一直下一步即可。

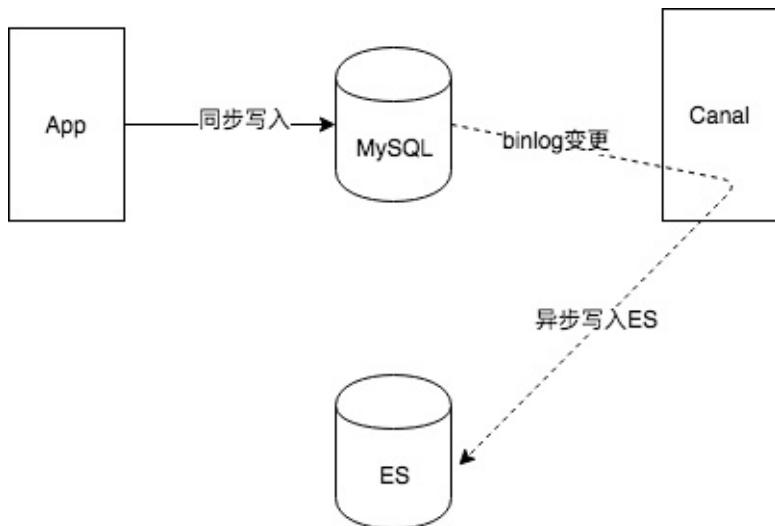
配置完成后，点击 Discover 菜单，就可看到程序DEMO打印的日志，并可进行搜索。现在，就请尽情探索吧。

实战：多数据源同步

通常情况下，使用ES的地方一般都会使用MySQL，将MySQL作为数据源，ES作为前台搜索。写入一条数据到MySQL时，也需要同时写入ES中。此时写入ES就有两种方式：一种同步的方式，另一种异步的方式。写入MySQL后同步写入ES，好处是实时更新，插入成功即可搜索，缺点也很明显，事务的问题(MySQL成功，ES失败的情况应提供一种保障机制达到数据一致性)，性能的问题(一条数据需要同时插入MySQL和ES成功后才能返回结果)。另一种情况是在写入MySQL成功后，异步写入ES，优点是数据一致性问题比较容易保证，性能的问题也不必等待过久，缺点也很明显，插入MySQL成功后，由于异步的原因，并不能立刻从ES搜索出结果。

写入MySQL成功，即数据库中数据新增了一条数据，利用MySQL的 binlog 技术能监测到数据的变化，从而发送一条MQ写入到ES中。有关 binlog 可自行搜索。Canal 即是[基于数据库增量日志解析，提供增量数据订阅和消费的一款阿里巴巴开源软件](#)。

在网络上已经有很多资料讲解如何通过 Canal 进行多数据源同步，这里不再给出详细的部署过程，下面是异步写入ES的架构图。



Canal下载地址：<https://github.com/alibaba/canal/releases>(下载deployer版本)。