第一节 ElasticSearch概述

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RESTfulweb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。构建在全 文检索开源软件Lucene之上的Elasticsearch，不仅能对海量规模的数据完成分布式索引与检索，还能提供数据聚合分析。据国际权威的 数据库产品评测机构DBEngines的统计，在2016年1月，Elasticsearch已超过Solr等，成为排名第一的搜索引擎类应用 概括：基于Restful标准的高扩展高可用的实时数据分析的全文搜索工具

1.2 ElasticSearch的基本概念

Index

类似于mysql数据库中的database

Type

类似于mysql数据库中的table表，es中可以在Index中建立type（table），通过mapping进行映射。

Document

由于es存储的数据是文档型的，一条数据对应一篇文档即相当于mysql数据库中的一行数据row， 一个文档中可以有多个字段也就是mysql数据库一行可以有多列。

Field

es中一个文档中对应的多个列与mysql数据库中每一列对应

Mapping

可以理解为mysql或者solr中对应的schema，只不过有些时候es中的mapping增加了动态识别功能，其实实际生产环境上不建议使用，最好还是开始制定好了对应的schema为主。

indexed

就是名义上的建立索引。mysql中一般会对经常使用的列增加相应的索引用于提高查询速度，而在es中默认都是会加 上索引的，除非你特殊制定不建立索引只是进行存储用于展示，这个需要看你具体的需求和业务进行设定了。

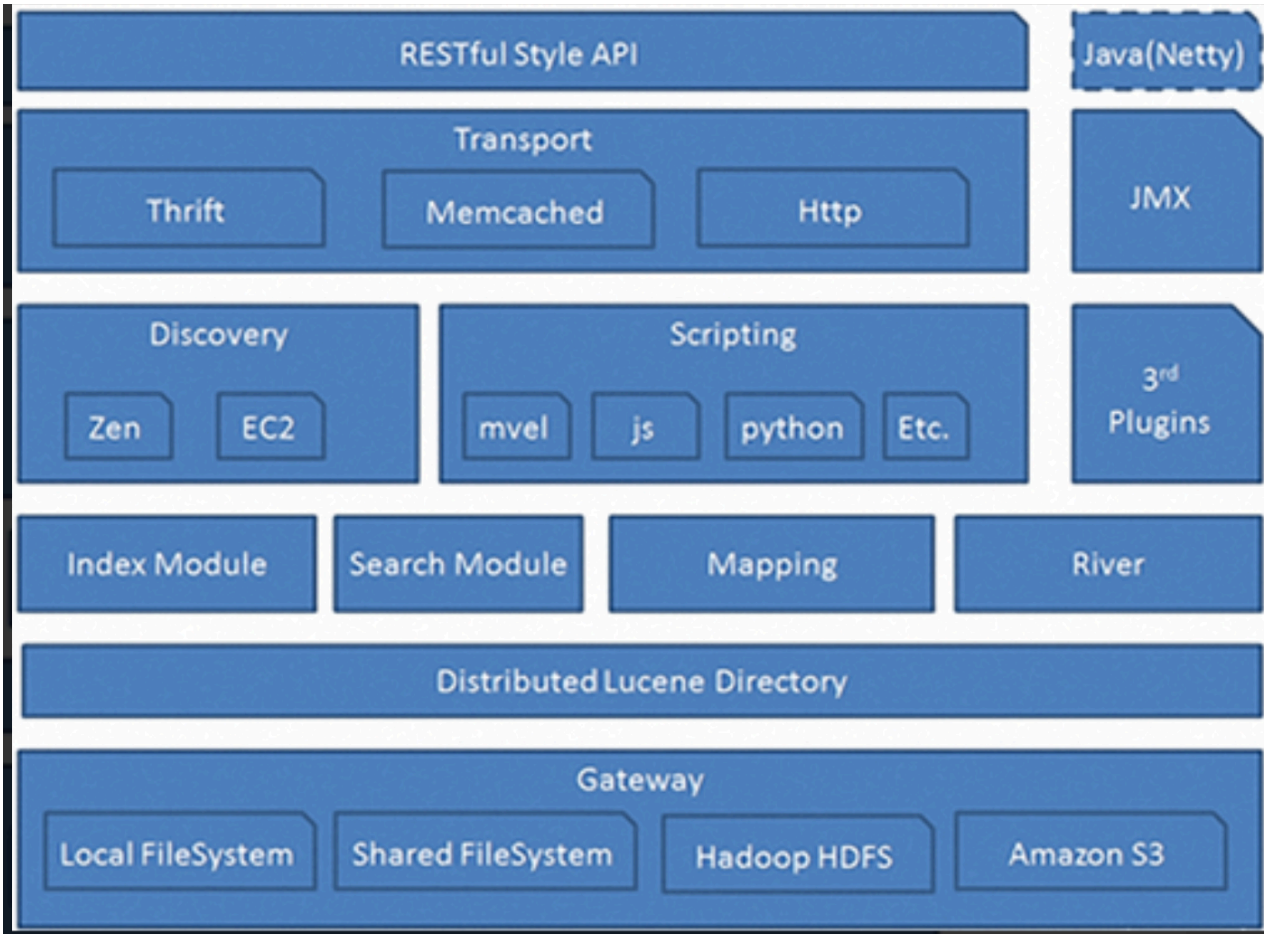
Query DSL

类似于mysql的sql语句，只不过在es中是使用的json格式的查询语句，专业术语就叫：QueryDSL

GET/PUT/POST/DELETE

分别类似与mysql中的select/update/delete……

1.3 Elasticsearch的架构



Gateway层

es用来存储索引文件的一个文件系统且它支持很多类型，例如：本地磁盘、共享存储（做snapshot的时候需要用到）、hadoop 的hdfs分布式存储、亚马逊的S3。它的主要职责是用来对数据进行长持久化以及整个集群重启之后可以通过gateway重新恢复数据。

Distributed Lucene Directory

Gateway上层就是一个lucene的分布式框架，lucene是做检索的，但是它是一个单机的搜索引擎，像这种es分布式搜索引擎系 统，虽然底层用lucene，但是需要在每个节点上都运行lucene进行相应的索引、查询以及更新，所以需要做成一个分布式的运 行框架来满足业务的需要。

四大模块组件

districted lucene directory之上就是一些es的模块

1.**Index Module**是索引模块，就是对数据建立索引也就是通常所说的建立一些倒排索引等；

2.**Search Module**是搜索模块，就是对数据进行查询搜索；

3.**Mapping**模块是数据映射与解析模块，就是你的数据的每个字段可以根据你建立的表结构 通过mapping进行映射解析，如果你没有建立表结构，es就会根据你的数据类型推测你 的数据结构之后自己生成一个mapping，然后都是根据这个mapping进行解析你的数据；

4.**River**模块在es2.0之后应该是被取消了，它的意思表示是第三方插件，例如可以通过一 些自定义的脚本将传统的数据库（mysql）等数据源通过格式化转换后直接同步到es集群里， 这个river大部分是自己写的，写出来的东西质量参差不齐，将这些东西集成到es中会引发 很多内部bug，严重影响了es的正常应用，所以在es2.0之后考虑将其去掉。

Discovery、Script

es4大模块组件之上有 Discovery模块：es是一个集群包含很多节点，很多节点需要互相发现对方，然后组成一个集群包括选 主的，这些es都是用的discovery模块，默认使用的是Zen，也可是使用EC2；es查询还可以支撑多种script即脚本语言，包括 mvel、js、python等等。

Transport协议层

再上一层就是es的通讯接口Transport，支持的也比较多：Thrift、Memcached以及Http，默认的是http，JMX就是java的一个 远程监控管理框架，因为es是通过java实现的。

RESTful接口层

最上层就是es暴露给我们的访问接口，官方推荐的方案就是这种Restful接口，直接发送http请求，方便后续使用nginx做代理、 分发包括可能后续会做权限的管理，通过http很容易做这方面的管理。如果使用java客户端它是直接调用api，在做负载均衡以 及权限管理还是不太好做。

1.4 RESTful API

一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。 基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。在目前主流的三种Web服务交互方案中，REST相比于S OAP（Simple Object Access protocol，简单对象访问协议）以及XML-RPC更加简单明了

```
(Representational State Transfer 意思是：表述性状态传递)

它使用典型的HTTP方法，诸如GET,POST.DELETE,PUT来实现资源的获取，添加，修改，删除等操作。即通过HTTP动词来实现资源的状态扭转


GET 用来获取资源

POST 用来新建资源（也可以用于更新资源）

PUT 用来更新资源

DELETE 用来删除资源
```

1.5 CRUL命令

```
以命令的方式执行HTTP协议的请求
GET/POST/PUT/DELETE

示例：
访问一个网页

curl www.baidu.com
```

```
curl -o tt.html www.baidu.com

显示响应的头信息

curl -i www.baidu.com

显示一次HTTP请求的通信过程

curl -v www.baidu.com

执行GET/POST/PUT/DELETE操作

curl -X GET/POST/PUT/DELETE url
```

第二节 ElasticSearch基本操作

2.1倒排索引

Elasticsearch 使用一种称为 倒排索引 的结构，它适用于快速的全文搜索。一个倒排索引由文档中所有不重复词的列表构成，对于其中每个词，有一个包含它的文档列表。

示例：

(1)：假设文档集合包含五个文档，每个文档内容如图所示，在图中最左端一栏是每个文档对应的文档编号。我们的任务就是对这个文档集合建立倒排索引。

(2):中文和英文等语言不同，单词之间没有明确分隔符号，所以首先要用分词系统将文档自动切分成单词序列。这样每个文档就转换为由单词序列构成的数据流，为了系统后续处理方便，需要对每个不同的单词赋予唯一的单词编号，同时记录下哪些文档包含这个单词，在如此处理结束后，我们可以得到最简单的倒排索引

“单词ID”一栏记录了每个单词的单词编号，第二栏是对应的单词，第三栏即每个单词对应的倒排列表

(3):索引系统还可以记录除此之外的更多信息,下图还记载了单词频率信息（TF）即这个单词在某个文档中的出现次数，之所以要记录这个信息，是因为词频信息在搜索结果排序时，计算查询和文档相似度是很重要的一个计算因子，所以将其记录在倒排列表中，以方便后续排序时进行分值计算。

单词ID	单词	倒排列表（DocID）
1	谷歌	1,2,3,4,5
2	地图	1,2,3,4,5
3	之父	1,2,4,5
4	跳槽	1,4
5	Facebook	1,2,3,4,5
6	加盟	2,3,5
7	创始人	3
8	拉斯	3,5
9	离开	3
10	与	4
11	Wave	4
12	项目	4
13	取消	4
14	有关	4
15	社交	5
16	网站	5

(4):倒排列表中还可以记录单词在某个文档出现的位置信息

(1,<11>,1),(2,<7>,1),(3,<3,9>,2)

有了这个索引系统，搜索引擎可以很方便地响应用户的查询，比如用户输入查询词“Facebook”，搜索系统查找倒排索引，从中可以读出包含这个单词的文档，这些文档就是提供给用户的搜索结果，而利用单词频率信息、文档频率信息即可以对这些候选搜索结果进行排序，计算文档和查询的相似性，按照相似性得分由高到低排序输出，此即为搜索系统的部分内部流程。

2.1.2 倒排索引原理

1.The quick brown fox jumped over the lazy dog

2.Quick brown foxes leap over lazy dogs in summer

倒排索引：

Term	**Doc_1**	**Doc_2**
Quick		X
The	X	
brown	X	X
dog	X	
dogs		X
fox	X	
foxes		X
in		X
jumped	X	
lazy	X	X
leap		X
over	X	X
quick	X	
summer		X
the	X	

搜索quick brown：

Term	**Doc_1**	**Doc_2**
brown	X	X
quick	X	
Total	2	1

计算相关度分数时，文档1的匹配度高，分数会比文档2高

问题：

Quick 和 quick 以独立的词条出现，然而用户可能认为它们是相同的词。

fox 和 foxes 非常相似, 就像 dog 和 dogs ； 他们有相同的词根。

jumped 和 leap, 尽管没有相同的词根，但他们的意思很相近。他们是同义词。

搜索含有 Quick fox的文档是搜索不到的

使用标准化规则(normalization)： 建立倒排索引的时候，会对拆分出的各个单词进行相应的处理，以提升后面搜索的时候能够搜索到相关联的文档的概率

Term	Doc_1	Doc_2
brown	X	X
dog	X	X
fox	X	X
in		X
jump	X	X
lazy	X	X
over	X	X
quick	X	X
summer		X
the	X	X

2.1.3 分词器介绍及内置分词器

分词器：从一串文本中切分出一个一个的词条，并对每个词条进行标准化

包括三部分：

character filter：分词之前的预处理，过滤掉HTML标签，特殊符号转换等

tokenizer：分词

token filter：标准化

内置分词器：

standard 分词器：(默认的)他会将词汇单元转换成小写形式，并去除停用词和标点符号，支持中文采用的方法为单字切分

simple 分词器：首先会通过非字母字符来分割文本信息，然后将词汇单元统一为小写形式。该分析器会去掉数字类型的字符。

Whitespace 分词器：仅仅是去除空格，对字符没有lowcase化,不支持中文； 并且不对生成的词汇单元进行其他的标准化处理。

language 分词器：特定语言的分词器，不支持中文

2.2 使用ElasticSearch API 实现CRUD

添加索引：

```
PUT /lib/

{

  "settings":{

      "index":{

        "number_of_shards": 5,

        "number_of_replicas": 1

        }

      }
}

PUT  lib
```

查看索引信息:

```
GET /lib/_settings

GET _all/_settings
```

添加文档:

```
PUT /lib/user/1

{
    "first_name" :  "Jane",

    "last_name" :   "Smith",

    "age" :         32,

    "about" :       "I like to collect rock albums",
```

```
    "interests":  [ "music" ]
}

POST /lib/user/

{
    "first_name" :  "Douglas",

    "last_name" :   "Fir",

    "age" :         23,

    "about":        "I like to build cabinets",

    "interests":  [ "forestry" ]

}
```

查看文档:

```
GET /lib/user/1

GET /lib/user/

GET /lib/user/1?_source=age,interests
```

更新文档:

```
PUT /lib/user/1

{
    "first_name" :  "Jane",

    "last_name" :   "Smith",

    "age" :         36,

    "about" :       "I like to collect rock albums",

    "interests":  [ "music" ]
}

POST /lib/user/1/_update

{
```

```
  "doc":{
      "age":33
      }
}
```

删除一个文档

```
DELETE /lib/user/1
```

删除一个索引

```
DELETE /lib
```

2.3 批量获取文档

使用es提供的Multi Get API：

使用Multi Get API可以通过索引名、类型名、文档id一次得到一个文档集合，文档可以来自同一个索引库，也可以来自不同索引库

使用curl命令：

```
curl 'http://192.168.25.131:9200/_mget' -d '{
"docs": [
   {
    "_index": "lib",
    "_type": "user",
    "_id": 1
   },
   {
     "_index": "lib",
     "_type": "user",
```

```
     "_id": 2

   }

  ]
}'
```

在客户端工具中：

```
GET /_mget

{

    "docs":[

       {
            "_index": "lib",
            "_type": "user",
            "_id": 1
       },
       {
            "_index": "lib",
            "_type": "user",
            "_id": 2
       },
       {
            "_index": "lib",
            "_type": "user",
            "_id": 3
       }

     ]
}
```

可以指定具体的字段：

```
GET /_mget

{

    "docs":[

       {
            "_index": "lib",
            "_type": "user",
            "_id": 1,
```

```
            "_source": "interests"
        },
        {
            "_index": "lib",
            "_type": "user",
            "_id": 2,
            "_source": ["age","interests"]
        }

    ]
}
```

获取同索引同类型下的不同文档：

```
GET /lib/user/_mget

{

    "docs":[

        {
            "_id": 1
        },
        {
            "_type": "user",
            "_id": 2,
        }

    ]
}

GET /lib/user/_mget

{

   "ids": ["1","2"]

}
```

2.4 使用Bulk API 实现批量操作

bulk的格式：

```
{action:{metadata}}\n

{requstbody}\n
```

```
action:(行为)

  create：文档不存在时创建

  update:更新文档

  index:创建新文档或替换已有文档

  delete:删除一个文档

metadata：_index,_type,_id
```

create 和index的区别

如果数据存在，使用create操作失败，会提示文档已经存在，使用index则可以成功执行。

示例：

```
{"delete":{"_index":"lib","_type":"user","_id":"1"}}
```

批量添加:

```
POST /lib2/books/_bulk

{"index":{"_id":1}}

{"title":"Java","price":55}

{"index":{"_id":2}}

{"title":"Html5","price":45}

{"index":{"_id":3}}

{"title":"Php","price":35}

{"index":{"_id":4}}

{"title":"Python","price":50}
```

批量获取:

```
GET /lib2/books/_mget
{

"ids": ["1","2","3","4"]
}
```

删除：没有请求体

```
POST /lib2/books/_bulk

{"delete":{"_index":"lib2","_type":"books","_id":4}}

{"create":{"_index":"tt","_type":"ttt","_id":"100"}}

{"name":"lisi"}

{"index":{"_index":"tt","_type":"ttt"}}

{"name":"zhaosi"}

{"update":{"_index":"lib2","_type":"books","_id":"4"}}

{"doc":{"price":58}}
```

bulk一次最大处理多少数据量:

bulk会把将要处理的数据载入内存中，所以数据量是有限制的，最佳的数据量不是一个确定的数值，它取决于你的硬件，你的文档大小以及复杂性，你的索引以及搜索的负载。

一般建议是1000-5000个文档，大小建议是5-15MB，默认不能超过100M，可以在es的配置文件（即$ES_HOME下的config下的elasticsearch.yml）中。

2.5 版本控制

ElasticSearch采用了乐观锁来保证数据的一致性，也就是说，当用户对document进行操作时，并不需要对该document作加锁和解锁的操作，只需要指定要操作的版本即可。当版本号一致时，ElasticSearch会允许该操作顺利执行，而当版本号存在冲突时，ElasticSearch会提示冲突并抛出异常（VersionConflictEngineException异常）。

ElasticSearch的版本号的取值范围为1到2^63-1。

内部版本控制：使用的是_version

外部版本控制：elasticsearch在处理外部版本号时会与对内部版本号的处理有些不同。它不再是检查*version是否与请求中指定的数值*相同,*而是检查当前的*version是否比指定的数值小。如果请求成功，那么外部的版本号就会被存储到文档中的_version中。

为了保持_version与外部版本控制的数据一致 使用version_type=external

2.6 什么是Mapping

```
PUT /myindex/article/1
{
  "post_date": "2018-05-10",
  "title": "Java",
  "content": "java is the best language",
  "author_id": 119
}

PUT /myindex/article/2
{
  "post_date": "2018-05-12",
  "title": "html",
  "content": "I like html",
  "author_id": 120
}

PUT /myindex/article/3
{
  "post_date": "2018-05-16",
  "title": "es",
  "content": "Es is distributed document store",
  "author_id": 110
}


GET /myindex/article/_search?q=2018-05

GET /myindex/article/_search?q=2018-05-10

GET /myindex/article/_search?q=html

GET /myindex/article/_search?q=java

#查看es自动创建的mapping

GET /myindex/article/_mapping
```

es自动创建了index，type，以及type对应的mapping(dynamic mapping)

什么是映射：mapping定义了type中的每个字段的数据类型以及这些字段如何分词等相关属性

```
{
  "myindex": {
    "mappings": {
      "article": {
```

```
        "properties": {
          "author_id": {
            "type": "long"
          },
          "content": {
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256
              }
            }
          },
          "post_date": {
            "type": "date"
          },
          "title": {
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256
              }
            }
          }
        }
      }
    }
  }
}
```

创建索引的时候,可以预先定义字段的类型以及相关属性，这样就能够把日期字段处理成日期，把数字字段处理成数字，把字符串字段处理字符串值等

支持的数据类型：

(1)核心数据类型（Core datatypes）

字符型：string，string类型包括
text 和 keyword

text类型被用来索引长文本，在建立索引前会将这些文本进行分词，转化为词的组合，建立索引。允许es来检索这些词语。text类型不能用来排序和聚合。

Keyword类型不需要进行分词，可以被用来检索过滤、排序和聚合。keyword 类型字段只能用本身来进行检索

数字型：long, integer, short, byte, double, float
日期型：date
布尔型：boolean
二进制型：binary

(2)复杂数据类型（Complex datatypes）

数组类型（Array datatype）：数组类型不需要专门指定数组元素的type，例如：
    字符型数组：["one", "two"]
    整型数组：[1, 2]
    数组型数组：[1, [2, 3]] 等价于[1, 2, 3]
    对象数组：[{ "name": "Mary", "age": 12 }, { "name": "John", "age": 10 }]
对象类型（Object datatype）：_ object _ 用于单个JSON对象；
嵌套类型（Nested datatype）：_ nested _ 用于JSON数组；

(3)地理位置类型（Geo datatypes）

地理坐标类型（Geo-point datatype）：_ geo_point _ 用于经纬度坐标；
地理形状类型（Geo-Shape datatype）：_ geo_shape _ 用于类似于多边形的复杂形状；

(4)特定类型（Specialised datatypes）

IPv4 类型（IPv4 datatype）：_ ip _ 用于IPv4 地址；
Completion 类型（Completion datatype）：_ completion _提供自动补全建议；
Token count 类型（Token count datatype）：_ token_count _ 用于统计做了标记的字段的index数目，该值会一直增加，不会因为过滤条件而减少。
mapper-murmur3
类型：通过插件，可以通过 _ murmur3 _ 来计算 index 的 hash 值；
附加类型（Attachment datatype）：采用 mapper-attachments
插件，可支持_ attachments _ 索引，例如 Microsoft Office 格式，Open Document 格式，ePub，HTML 等。

支持的属性：

"store":false//是否单独设置此字段的是否存储而从_source字段中分离，默认是false，只能搜索，不能获取值

"index": true//分词，不分词是：false ，设置成false，字段将不会被索引

"analyzer":"ik"//指定分词器,默认分词器为standard analyzer

"boost":1.23//字段级别的分数加权，默认值是1.0

"doc_values":false//对not_analyzed字段，默认都是开启，分词字段不能使用，对排序和聚合能提升较大性能，节约内存

"fielddata":{"format":"disabled"}//针对分词字段，参与排序或聚合时能提高性能，不分词字段统一建议使用doc_value

"fields":{"raw":{"type":"string","index":"not_analyzed"}} //可以对一个字段提供多种索引模式，同一个字段的值，一个分词，一个不分词

"ignore_above":100 //超过100个字符的文本，将会被忽略，不被索引

"include_in_all":ture//设置是否此字段包含在_all字段中，默认是true，除非index设置成no选项

"index_options":"docs"//4个可选参数docs（索引文档号） ,freqs（文档号+词频），positions（文档号+词频+位置，通常用来距离查询），offsets（文档号+词频+位置+偏移量，通常被使用在高亮字段）分词字段默认是position，其他的默认是docs

"norms":{"enable":true,"loading":"lazy"}//分词字段默认配置，不分词字段：默认{"enable":false}，存储长度因子和索引时boost，建议对需要参与评分字段使用 ，会额外增加内存消耗量

"null_value":"NULL"//设置一些缺失字段的初始化值，只有string可以使用，分词字段的null值也会被分词

"position_increament_gap":0//影响距离查询或近似查询，可以设置在多值字段的数据上火分词字段上，查询时可指定slop间隔，默认值是100

"search_analyzer":"ik"//设置搜索时的分词器，默认跟ananlyzer是一致的，比如index时用standard+ngram，搜索时用standard用来完成自动提示功能

"similarity":"BM25"//默认是TF/IDF算法，指定一个字段评分策略，仅仅对字符串型和分词类型有效

"term_vector":"no"//默认不存储向量信息，支持参数yes（term存储），with_positions（term+位置）,with_offsets（term+偏移量），with_positions_offsets(term+位置+偏移量) 对快速高亮fast vector highlighter能提升性能，但开启又会加大索引体积，不适合大数据量用

映射的分类：

(1)动态映射：

当ES在文档中碰到一个以前没见过的字段时，它会利用动态映射来决定该字段的类型，并自动地对该字段添加映射。

可以通过dynamic设置来控制这一行为，它能够接受以下的选项：

```
true：默认值。动态添加字段
false：忽略新字段
strict：如果碰到陌生字段，抛出异常
```

dynamic设置可以适用在根对象上或者object类型的任意字段上。

POST /lib2

#给索引lib2创建映射类型

```
{

    "settings":{

    "number_of_shards" : 3,

    "number_of_replicas" : 0

    },

     "mappings":{

       "books":{

         "properties":{

             "title":{"type":"text"},
             "name":{"type":"text","index":false},
             "publish_date":{"type":"date","index":false},

             "price":{"type":"double"},

             "number":{"type":"integer"}
         }
       }
      }
}
```

POST /lib2

#给索引lib2创建映射类型

```
{

    "settings":{

    "number_of_shards" : 3,
```

```
    "number_of_replicas" : 0

    },

     "mappings":{

      "books":{

        "properties":{

            "title":{"type":"text"},
            "name":{"type":"text","index":false},
            "publish_date":{"type":"date","index":false},

            "price":{"type":"double"},

            "number":{
                "type":"object",
                "dynamic":true
            }
        }
      }
     }
}
```

2.7 基本查询(Query查询)

2.7.1 数据准备

```
PUT /lib3
{
    "settings":{
    "number_of_shards" : 3,
    "number_of_replicas" : 0
    },
     "mappings":{
      "user":{
        "properties":{
            "name": {"type":"text"},
            "address": {"type":"text"},
            "age": {"type":"integer"},
            "interests": {"type":"text"},
            "birthday": {"type":"date"}
        }
      }
     }
}
```

```
GET /lib3/user/_search?q=name:lisi

GET /lib3/user/_search?q=name:zhaoliu&sort=age:desc
```

2.7.2 term查询和terms查询

term query会去倒排索引中寻找确切的term，它并不知道分词器的存在。这种查询适合keyword 、numeric、date。

term:查询某个字段里含有某个关键词的文档

```
GET /lib3/user/_search/
{
  "query": {
      "term": {"interests": "changge"}
  }
}
```

terms:查询某个字段里含有多个关键词的文档

```
GET /lib3/user/_search
{
    "query":{
        "terms":{
            "interests": ["hejiu","changge"]
        }
    }
}
```

2.7.3 控制查询返回的数量

from：从哪一个文档开始 size：需要的个数

```
GET /lib3/user/_search
{
    "from":0,
    "size":2,
    "query":{
        "terms":{
            "interests": ["hejiu","changge"]
        }
    }
}
```

2.7.4 返回版本号

```
GET /lib3/user/_search
{
    "version":true,
    "query":{
        "terms":{
            "interests": ["hejiu","changge"]
        }
    }
}
```

2.7.5 match查询

match query知道分词器的存在，会对filed进行分词操作，然后再查询

```
GET /lib3/user/_search
{
    "query":{
        "match":{
            "name": "zhaoliu"
        }
    }
}
GET /lib3/user/_search
{
    "query":{
        "match":{
            "age": 20
        }
    }
}
```

match_all:查询所有文档

```
GET /lib3/user/_search
{
  "query": {
    "match_all": {}
  }
}
```

multi_match:可以指定多个字段

```
GET /lib3/user/_search
{
    "query":{
        "multi_match": {
            "query": "lvyou",
            "fields": ["interests","name"]
         }
    }
}
```

match_phrase:短语匹配查询

ElasticSearch引擎首先分析（analyze）查询字符串，从分析后的文本中构建短语查询，这意味着必须匹配短语中的所有分词，并且保证各个分词的相对位置不变：

```
GET lib3/user/_search
{
  "query":{
      "match_phrase":{
         "interests": "duanlian, shuoxiangsheng"
      }
   }
}
```

2.7.6 指定返回的字段

```
GET /lib3/user/_search
{
    "_source": ["address","name"],
    "query": {
        "match": {
            "interests": "changge"
        }
    }
}
```

2.7.7 控制加载的字段

```
GET /lib3/user/_search
{
    "query": {
        "match_all": {}
    },

    "_source": {
          "includes": ["name","address"],
          "excludes": ["age","birthday"]
      }
}
```

使用通配符*

```
GET /lib3/user/_search
{
    "_source": {
          "includes": "addr*",
          "excludes": ["name","bir*"]

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

2.7.8 排序

使用sort实现排序： desc:降序，asc升序

```
GET /lib3/user/_search
{
    "query": {
        "match_all": {}
    },
    "sort": [
        {
           "age": {
               "order":"asc"
           }
        }
    ]

}

GET /lib3/user/_search
```

```
{
    "query": {
        "match_all": {}
    },
    "sort": [
        {
           "age": {
               "order":"desc"
           }
        }
    ]

}
```

2.7.9 前缀匹配查询

```
GET /lib3/user/_search
{
  "query": {
    "match_phrase_prefix": {
        "name": {
            "query": "zhao"
        }
    }
  }
}
```

2.7.10 范围查询

range:实现范围查询

参数：from,to,include_lower,include_upper,boost

include_lower:是否包含范围的左边界，默认是true

include_upper:是否包含范围的右边界，默认是true

```
GET /lib3/user/_search
{
    "query": {
        "range": {
            "birthday": {
                "from": "1990-10-10",
                "to": "2018-05-01"
            }
        }
    }
}
```

```
GET /lib3/user/_search
{
    "query": {
        "range": {
            "age": {
                "from": 20,
                "to": 25,
                "include_lower": true,
                "include_upper": false
            }
        }
    }
}
```

2.7.11 wildcard查询

允许使用通配符* 和 ?来进行查询

*代表0个或多个字符

? 代表任意一个字符

```
GET /lib3/user/_search
{
    "query": {
        "wildcard": {
             "name": "zhao*"
        }
    }
}


GET /lib3/user/_search
{
    "query": {
        "wildcard": {
             "name": "li?i"
        }
    }
}
```

2.7.12 fuzzy实现模糊查询

value：查询的关键字

boost：查询的权值，默认值是1.0

min_similarity:设置匹配的最小相似度，默认值为0.5，对于字符串，取值为0-1(包括0和1);对于数值，取值可能大于1;对于日期型取值为1d,1m等，1d就代表1天

prefix_length:指明区分词项的共同前缀长度，默认是0

max_expansions:查询中的词项可以扩展的数目，默认可以无限大

```
GET /lib3/user/_search
{
    "query": {
        "fuzzy": {
             "interests": "chagge"
        }
    }
}

GET /lib3/user/_search
{
    "query": {
        "fuzzy": {
             "interests": {
                 "value": "chagge"
             }
        }
    }
}
```

2.7.13 高亮搜索结果

```
GET /lib3/user/_search
{
    "query":{
        "match":{
            "interests": "changge"
        }
    },
    "highlight": {
        "fields": {
             "interests": {}
        }
    }
}
```

2.8 Filter查询

filter是不计算相关性的，同时可以cache。因此，filter速度要快于query。

```
POST /lib4/items/_bulk
{"index": {"_id": 1}}

{"price": 40,"itemID": "ID100123"}

{"index": {"_id": 2}}

{"price": 50,"itemID": "ID100124"}

{"index": {"_id": 3}}

{"price": 25,"itemID": "ID100124"}

{"index": {"_id": 4}}

{"price": 30,"itemID": "ID100125"}

{"index": {"_id": 5}}

{"price": null,"itemID": "ID100127"}
```

####2.8.1 简单的过滤查询

```
GET /lib4/items/_search
{
       "post_filter": {
             "term": {
                 "price": 40
             }
       }
}


GET /lib4/items/_search
{
      "post_filter": {
          "terms": {
                 "price": [25,40]
              }
        }
}

GET /lib4/items/_search
{
    "post_filter": {
        "term": {
            "itemID": "ID100123"
          }
```

```
      }
}
```

查看分词器分析的结果：

```
GET /lib4/_mapping
```

不希望商品id字段被分词，则重新创建映射

```
DELETE lib4

PUT /lib4
{
    "mappings": {
        "items": {
            "properties": {
                "itemID": {
                    "type": "text",
                    "index": false
                }
            }
        }
    }
}
```

2.8.2 bool过滤查询

可以实现组合过滤查询

格式：

```
{
    "bool": {
        "must": [],
        "should": [],
        "must_not": []
    }
}
```

must:必须满足的条件---and

should：可以满足也可以不满足的条件--or

must_not:不需要满足的条件--not

```
GET /lib4/items/_search
{
    "post_filter": {
          "bool": {
               "should": [
                    {"term": {"price":25}},
                    {"term": {"itemID": "id100123"}}

                 ],
                "must_not": {
                    "term":{"price": 30}
                  }

                }
             }
}
```

嵌套使用bool：

```
GET /lib4/items/_search
{
    "post_filter": {
          "bool": {
                "should": [
                    {"term": {"itemID": "id100123"}},
                    {
                      "bool": {
                          "must": [
                              {"term": {"itemID": "id100124"}},
                              {"term": {"price": 40}}
                            ]
                          }
                    }
                  ]
                }
            }
}
```

2.8.3 范围过滤

gt: >

lt: <

gte: >=

lte: <=

```
GET /lib4/items/_search
{
     "post_filter": {
          "range": {
              "price": {
                   "gt": 25,
                   "lt": 50
                }
            }
      }
}
```

2.8.5 过滤非空

```
GET /lib4/items/_search
{
  "query": {
    "bool": {
      "filter": {
          "exists":{
             "field":"price"
         }
      }
    }
  }
}

GET /lib4/items/_search
{
    "query" : {
        "constant_score" : {
            "filter": {
                "exists" : { "field" : "price" }
            }
        }
    }
}
```

2.8.6 过滤器缓存

ElasticSearch提供了一种特殊的缓存，即过滤器缓存（filter cache），用来存储过滤器的结果，被缓存的过滤器并不需要消耗过多的内存（因为它们只存储了哪些文档能与过滤器相匹配的相关信息），而且可供后续所有与之相关的查询重复使用，从而极大地提高了查询性能。

注意：ElasticSearch并不是默认缓存所有过滤器， 以下过滤器默认不缓存：

```
numeric_range
script
geo_bbox
geo_distance
geo_distance_range
geo_polygon
geo_shape
and
or
not
```

exists,missing,range,term,terms默认是开启缓存的

开启方式：在filter查询语句后边加上 "_catch":true

2.9 聚合查询

(1)sum

```
GET /lib4/items/_search
{
  "size":0,
  "aggs": {
     "price_of_sum": {
         "sum": {
           "field": "price"
         }
     }
  }
}
```

(2)min

```
GET /lib4/items/_search
{
  "size": 0,
  "aggs": {
     "price_of_min": {
         "min": {
           "field": "price"
         }
     }
  }
}
```

(3)max

```
GET /lib4/items/_search
{
  "size": 0,
  "aggs": {
     "price_of_max": {
         "max": {
           "field": "price"
         }
     }
  }
}
```

(4)avg

```
GET /lib4/items/_search
{
  "size":0,
  "aggs": {
     "price_of_avg": {
         "avg": {
           "field": "price"
         }
     }
  }
}
```

(5)cardinality:求基数

```
GET /lib4/items/_search
{
  "size":0,
  "aggs": {
     "price_of_cardi": {
         "cardinality": {
           "field": "price"
         }
     }
  }
}
```

(6)terms:分组

```
GET /lib4/items/_search
{
  "size":0,
  "aggs": {
     "price_group_by": {
         "terms": {
           "field": "price"
         }
     }
  }
}
```

对那些有唱歌兴趣的用户按年龄分组

```
GET /lib3/user/_search
{
  "query": {
      "match": {
        "interests": "changge"
      }
   },
   "size": 0,
   "aggs":{
       "age_group_by":{
           "terms": {
             "field": "age",
             "order": {
               "avg_of_age": "desc"
             }
           },
           "aggs": {
             "avg_of_age": {
               "avg": {
                 "field": "age"
               }
             }
           }
       }
   }
}
```

2.10 复合查询

将多个基本查询组合成单一查询的查询

2.10.1 使用bool查询

接收以下参数：

must： 文档 必须匹配这些条件才能被包含进来。

must_not： 文档 必须不匹配这些条件才能被包含进来。

should： 如果满足这些语句中的任意语句，将增加 _score，否则，无任何影响。它们主要用于修正每个文档的相关性得分。

filter： 必须 匹配，但它以不评分、过滤模式来进行。这些语句对评分没有贡献，只是根据过滤标准来排除或包含文档。

相关性得分是如何组合的。每一个子查询都独自地计算文档的相关性得分。一旦他们的得分被计算出来， bool 查询就将这些得分进行合并并且返回一个代表整个布尔操作的得分。

下面的查询用于查找 title 字段匹配 how to make millions 并且不被标识为 spam 的文档。那些被标识为 starred 或在2014之后的文档，将比另外那些文档拥有更高的排名。如果 *两者* 都满足，那么它排名将更高：

```
{
    "bool": {
        "must": { "match": { "title": "how to make millions" }},
        "must_not": { "match": { "tag":   "spam" }},
        "should": [
            { "match": { "tag": "starred" }},
            { "range": { "date": { "gte": "2014-01-01" }}}
        ]
    }
}
```

如果没有 must 语句，那么至少需要能够匹配其中的一条 should 语句。但，如果存在至少一条 must 语句，则对 should 语句的匹配没有要求。 如果我们不想因为文档的时间而影响得分，可以用 filter 语句来重写前面的例子：

```
{
    "bool": {
        "must": { "match": { "title": "how to make millions" }},
        "must_not": { "match": { "tag":   "spam" }},
        "should": [
            { "match": { "tag": "starred" }}
        ],
        "filter": {
          "range": { "date": { "gte": "2014-01-01" }}
        }
    }
}
```

通过将 range 查询移到 filter 语句中，我们将它转成不评分的查询，将不再影响文档的相关性排名。由于它现在是一个不评分的查询，可以使用各种对 filter 查询有效的优化手段来提升性能。

bool 查询本身也可以被用做不评分的查询。简单地将它放置到 filter 语句中并在内部构建布尔逻辑：

```
{
    "bool": {
        "must": { "match": { "title": "how to make millions" }},
        "must_not": { "match": { "tag":   "spam" }},
        "should": [
            { "match": { "tag": "starred" }}
        ],
        "filter": {
          "bool": {
              "must": [
                  { "range": { "date": { "gte": "2014-01-01" }}},
                  { "range": { "price": { "lte": 29.99 }}}
              ],
              "must_not": [
                  { "term": { "category": "ebooks" }}
              ]
          }
        }
    }
}
```

2.10.2 constant_score查询

它将一个不变的常量评分应用于所有匹配的文档。它被经常用于你只需要执行一个 filter 而没有其它查询（例如，评分查询）的情况下。

```
{
    "constant_score":   {
        "filter": {
            "term": { "category": "ebooks" }
        }
    }
}
```

term 查询被放置在 constant_score 中，转成不评分的filter。这种方式可以用来取代只有 filter 语句的 bool 查询。

第三节 ElasticSearch原理

3.1 解析es的分布式架构

3.1.1 分布式架构的透明隐藏特性

ElasticSearch是一个分布式系统，隐藏了复杂的处理机制

分片机制：我们不用关心数据是按照什么机制分片的、最后放入到哪个分片中

分片的副本：

集群发现机制(cluster discovery)：比如当前我们启动了一个es进程，当启动了第二个es进程时，这个进程作为一个node自动就发现了集群，并且加入了进去

shard负载均衡：比如现在有10shard，集群中有3个节点，es会进行均衡的进行分配，以保持每个节点均衡的负载请求

请求路由

3.1.2 扩容机制

垂直扩容：购置新的机器，替换已有的机器

水平扩容：直接增加机器

3.1.3 rebalance

增加或减少节点时会自动均衡

3.1.4 master节点

主节点的主要职责是和集群操作相关的内容，如创建或删除索引，跟踪哪些节点是群集的一部分，并决定哪些分片分配给相关的节点。稳定的主节点对集群的健康是非常重要的。

3.1.5 节点对等

每个节点都能接收请求 每个节点接收到请求后都能把该请求路由到有相关数据的其它节点上 接收原始请求的节点负责采集数据并返回给客户端

3.2 分片和副本机制

1.index包含多个shard

2.每个shard都是一个最小工作单元，承载部分数据；每个shard都是一个lucene实例，有完整的建立索引和处理请求的能力

3.增减节点时，shard会自动在nodes中负载均衡

4.primary shard和replica shard，每个document肯定只存在于某一个primary shard以及其对应的replica shard中，不可能存在于多个primary shard

5.replica shard是primary shard的副本，负责容错，以及承担读请求负载

6.primary shard的数量在创建索引的时候就固定了，replica shard的数量可以随时修改

7.primary shard的默认数量是5，replica默认是1，默认有10个shard，5个primary shard，5个replica shard

8.primary shard不能和自己的replica shard放在同一个节点上（否则节点宕机，primary shard和副本都丢失，起不到容错的作用），但是可以和其他primary shard的replica shard放在同一个节点上

3.3 单节点环境下创建索引分析

```
PUT /myindex
{
   "settings" : {
      "number_of_shards" : 3,
      "number_of_replicas" : 1
   }
}
```

这个时候，只会将3个primary shard分配到仅有的一个node上去，另外3个replica shard是无法分配的（一个shard的副本replica，他们两个是不能在同一个节点的）。集群可以正常工作，但是一旦出现节点宕机，数据全部丢失，而且集群不可用，无法接收任何请求。

3.4 两个节点环境下创建索引分析

将3个primary shard分配到一个node上去，另外3个replica shard分配到另一个节点上

primary shard 和replica shard 保持同步

primary shard 和replica shard 都可以处理客户端的读请求

3.5 水平扩容的过程

1.扩容后primary shard和replica shard会自动的负载均衡

2.扩容后每个节点上的shard会减少，那么分配给每个shard的CPU，内存，IO资源会更多，性能提高

3.扩容的极限，如果有6个shard，扩容的极限就是6个节点，每个节点上一个shard，如果想超出扩容的极限，比如说扩容到9个节点，那么可以增加replica shard的个数

4.6个shard，3个节点，最多能承受几个节点所在的服务器宕机？(容错性) 任何一台服务器宕机都会丢失部分数据

为了提高容错性，增加shard的个数： 9个shard，(3个primary shard，6个replicashard)，这样就能容忍最多两台服务器宕机了

总结：扩容是为了提高系统的吞吐量，同时也要考虑容错性，也就是让尽可能多的服务器宕机还能保证数据不丢失

3.6 ElasticSearch的容错机制

以9个shard，3个节点为例：

1.如果master node 宕机，此时不是所有的primary shard都是Active status，所以此时的集群状态是red。

容错处理的第一步:是选举一台服务器作为master 容错处理的第二步:新选举出的master会把挂掉的primary shard的某个replica shard 提升为primary shard,此时集群的状态为yellow，因为少了一个replica shard，并不是所有的replica shard都是active status

容错处理的第三步：重启故障机，新master会把所有的副本都复制一份到该节点上，（同步一下宕机后发生的修改），此时集群的状态为green，因为所有的primary shard和replica shard都是Active status

3.7 文档的核心元数据

1._index:

说明了一个文档存储在哪个索引中

同一个索引下存放的是相似的文档(文档的field多数是相同的)

索引名必须是小写的，不能以下划线开头，不能包括逗号

2._type:

表示文档属于索引中的哪个类型

一个索引下只能有一个type

类型名可以是大写也可以是小写的，不能以下划线开头，不能包括逗号

3._id:

文档的唯一标识，和索引，类型组合在一起唯一标识了一个文档

可以手动指定值，也可以由es来生成这个值

3.8 文档id生成方式

1.手动指定

```
put /index/type/66
```

通常是把其它系统的已有数据导入到es时

2.由es生成id值

```
post /index/type
```

es生成的id长度为20个字符，使用的是base64编码，URL安全，使用的是GUID算法，分布式下并发生成id值时不会冲突

3.9 _source元数据分析

其实就是我们在添加文档时request body中的内容

指定返回的结果中含有哪些字段：

```
get /index/type/1?_source=name
```

3.10 改变文档内容原理解析

替换方式：

```
PUT /lib/user/4
{ "first_name" : "Jane",

"last_name" :   "Lucy",

"age" :         24,

"about" :       "I like to collect rock albums",

"interests":  [ "music" ]
}
```

修改方式(partial update)：

```
POST /lib/user/2/_update
{
    "doc":{
       "age":26
     }
}
```

删除文档：标记为deleted，随着数据量的增加，es会选择合适的时间删除掉

3.11 基于groovy脚本执行partial update

es有内置的脚本支持，可以基于groovy脚本实现复杂的操作

1.修改年龄

```
POST /lib/user/4/_update
{
  "script": "ctx._source.age+=1"
}
```

2.修改名字

```
POST /lib/user/4/_update
{
  "script": "ctx._source.last_name+='hehe'"
}
```

3.添加爱好

```
POST /lib/user/4/_update
{
  "script": {
    "source": "ctx._source.interests.add(params.tag)",
    "params": {
      "tag":"picture"
    }
  }
}
```

4.删除爱好

```
POST /lib/user/4/_update
{
  "script": {
    "source":
"ctx._source.interests.remove(ctx._source.interests.indexOf(params.tag))",
    "params": {
      "tag":"picture"
    }
  }
}
```

5.删除文档

```
POST /lib/user/4/_update
{
  "script": {
    "source": "ctx.op=ctx._source.age==params.count?'delete':'none'",
    "params": {
        "count":29
    }
  }
}

6.upsert
```

```
POST /lib/user/4/_update
{
  "script": "ctx._source.age += 1",

  "upsert": {
     "first_name" : "Jane",
     "last_name" :   "Lucy",
     "age" :  20,
     "about" :       "I like to collect rock albums",
     "interests":  [ "music" ]
  }
}
```

3.12 partial update 处理并发冲突

使用的是乐观锁:_version

retry_on_conflict:

POST /lib/user/4/_update?retry_on_conflict=3

重新获取文档数据和版本信息进行更新，不断的操作，最多操作的次数就是retry_on_conflict的值

3.13 文档数据路由原理解析

1.文档路由到分片上：

一个索引由多个分片构成，当添加(删除，修改)一个文档时，es就需要决定这个文档存储在哪个分片上，这个过程就称为数据路由(routing)

2.路由算法：

```
shard=hash(routing) % number_of_pirmary_shards
```

示例：一个索引，3个primary shard

(1)每次增删改查时，都有一个routing值，默认是文档的_id的值

(2)对这个routing值使用哈希函数进行计算

(3)计算出的值再和主分片个数取余数

余数肯定在0---（number_of_pirmary_shards-1）之间，文档就在对应的shard上

routing值默认是文档的_id的值，也可以手动指定一个值，手动指定对于负载均衡以及提高批量读取的性能都有帮助

3.primary shard个数一旦确定就不能修改了

3.14 文档增删改内部原理

1:发送增删改请求时，可以选择任意一个节点，该节点就成了协调节点(coordinating node)

2.协调节点使用路由算法进行路由，然后将请求转到primary shard所在节点，该节点处理请求，并把数据同步到它的replica shard

3.协调节点对客户端做出响应

3.15 写一致性原理和quorum机制

1.任何一个增删改操作都可以跟上一个参数 consistency

可以给该参数指定的值：

one: (primary shard)只要有一个primary shard是活跃的就可以执行

all: (all shard)所有的primary shard和replica shard都是活跃的才能执行

quorum: (default) 默认值，大部分shard是活跃的才能执行 （例如共有6个shard，至少有3个shard是活跃的才能执行写操作）

2.quorum机制：多数shard都是可用的，

int((primary+number_of_replica)/2)+1

例如：3个primary shard，1个replica

int((3+1)/2)+1=3

至少3个shard是活跃的

注意：可能出现shard不能分配齐全的情况

比如：1个primary shard,1个replica int((1+1)/2)+1=2 但是如果只有一个节点，因为primary shard和replica shard不能在同一个节点上，所以仍然不能执行写操作

再举例：1个primary shard,3个replica,2个节点

int((1+3)/2)+1=3

最后:当活跃的shard的个数没有达到要求时， es默认会等待一分钟，如果在等待的期间活跃的shard的个数没有增加，则显示timeout

put /index/type/id?timeout=60s

3.16 文档查询内部原理

第一步：查询请求发给任意一个节点，该节点就成了coordinating node，该节点使用路由算法算出文档所在的primary shard

第二步：协调节点把请求转发给primary shard也可以转发给replica shard(使用轮询调度算法(Round-Robin Scheduling，把请求平均分配至primary shard 和replica shard)

第三步：处理请求的节点把结果返回给协调节点，协调节点再返回给应用程序

特殊情况：请求的文档还在建立索引的过程中，primary shard上存在，但replica shar上不存在，但是请求被转发到了replica shard上，这时就会提示找不到文档

3.17 bulk批量操作的json格式解析

bulk的格式：

```
{action:{metadata}}\n

{requstbody}\n
```

为什么不使用如下格式：

```
[{

"action": {

},

"data": {

}

}]
```

这种方式可读性好，但是内部处理就麻烦了：

1.将json数组解析为JSONArray对象，在内存中就需要有一份json文本的拷贝，另外还有一个JSONArray对象。

2.解析json数组里的每个json，对每个请求中的document进行路由

3.为路由到同一个shard上的多个请求，创建一个请求数组

4.将这个请求数组序列化

5.将序列化后的请求数组发送到对应的节点上去

耗费更多内存，增加java虚拟机开销

1.不用将其转换为json对象，直接按照换行符切割json，内存中不需要json文本的拷贝

2.对每两个一组的json，读取meta，进行document路由

3.直接将对应的json发送到node上去

3.18 查询结果分析

```
{
  "took": 419,
  "timed_out": false,
  "_shards": {
```

```
    "total": 3,
    "successful": 3,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 3,
    "max_score": 0.6931472,
    "hits": [
      {
        "_index": "lib3",
        "_type": "user",
        "_id": "3",
        "_score": 0.6931472,
        "_source": {
          "address": "bei jing hai dian qu qing he zhen",
          "name": "lisi"
        }
      },
      {
        "_index": "lib3",
        "_type": "user",
        "_id": "2",
        "_score": 0.47000363,
        "_source": {
          "address": "bei jing hai dian qu qing he zhen",
          "name": "zhaoming"
        }
      }
```


took：查询耗费的时间，单位是毫秒

_shards：共请求了多少个shard

total：查询出的文档总个数

max_score： 本次查询中，相关度分数的最大值，文档和此次查询的匹配度越高，_score的值越大，排位越靠前

hits：默认查询前10个文档

timed_out：

```
GET /lib3/user/_search?timeout=10ms
{
    "_source": ["address","name"],
    "query": {
        "match": {
            "interests": "changge"
        }
    }
}
```

3.19 多index，多type查询模式

```
GET _search

GET /lib/_search

GET /lib,lib3/_search

GET /*3,*4/_search

GET /lib/user/_search

GET /lib,lib4/user,items/_search

GET /_all/_search

GET /_all/user,items/_search
```

3.20 分页查询中的deep paging问题

```
GET /lib3/user/_search
{
    "from":0,
    "size":2,
    "query":{
        "terms":{
            "interests": ["hejiu","changge"]
        }
    }
}

GET /_search?from=0&size=3
```

deep paging:查询的很深，比如一个索引有三个primary shard，分别存储了6000条数据，我们要得到第100页的数据(每页10条)，类似这种情况就叫deep paging

如何得到第100页的10条数据?

在每个shard中搜索990到999这10条数据，然后用这30条数据排序，排序之后取10条数据就是要搜索的数据，这种做法是错的，因为3个shard中的数据的*score分数不一样，可能这某一个shard中第一条数据的*score分数比另一个shard中第1000条都要高，所以在每个shard中搜索990到999这10条数据然后排序的做法是不正确的。

正确的做法是每个shard把0到999条数据全部搜索出来（按排序顺序），然后全部返回给coordinate node，由coordinate node按_score分数排序后，取出第100页的10条数据，然后返回给客户端。

deep paging性能问题

1.耗费网络带宽，因为搜索过深的话，各shard要把数据传送给coordinate node，这个过程是有大量数据传递的，消耗网络，

2.消耗内存，各shard要把数据传送给coordinate node，这个传递回来的数据，是被coordinate node保存在内存中的，这样会大量消耗内存。

3.消耗cpu coordinate node要把传回来的数据进行排序，这个排序过程很消耗cpu.

鉴于deep paging的性能问题，所以应尽量减少使用。

3.21 query string查询及copy_to解析

```
GET /lib3/user/_search?q=interests:changge

GET /lib3/user/_search?q=+interests:changge

GET /lib3/user/_search?q=-interests:changge
```

copy_to字段是把其它字段中的值，以空格为分隔符组成一个大字符串，然后被分析和索引，但是不存储，也就是说它能被查询，但不能被取回显示。

注意:copy_to指向的字段字段类型要为：text

当没有指定field时，就会从copy_to字段中查询

```
GET /lib3/user/_search?q=changge
```

3.22 字符串排序问题

对一个字符串类型的字段进行排序通常不准确，因为已经被分词成多个词条了

解决方式：对字段索引两次，一次索引分词（用于搜索），一次索引不分词(用于排序)

```
GET /lib3/_search
```

```
GET /lib3/user/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "interests": {
        "order": "desc"
      }
    }
  ]
}

GET /lib3/user/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "interests.raw": {
        "order": "asc"
      }
    }
  ]
}

DELETE lib3

PUT /lib3
{
    "settings":{
        "number_of_shards" : 3,
        "number_of_replicas" : 0
      },
     "mappings":{
      "user":{
        "properties":{
            "name": {"type":"text"},
            "address": {"type":"text"},
            "age": {"type":"integer"},
            "birthday": {"type":"date"},
            "interests": {
                "type":"text",
                "fields": {
                  "raw":{
                     "type": "keyword"
                   }
```

```
                },
                "fielddata": true
            }
        }
    }
}
```

3.23 如何计算相关度分数

使用的是TF/IDF算法(Term Frequency&Inverse Document Frequency)

1.Term Frequency:我们查询的文本中的词条在document本中出现了多少次，出现次数越多，相关度越高

```
搜索内容： hello world

Hello, I love china.

Hello world,how are you!
```

2.Inverse Document Frequency：我们查询的文本中的词条在索引的所有文档中出现了多少次，出现的次数越多，相关度越低

```
搜索内容：hello world

hello, what are you doing?

I like the world.

hello 在索引的所有文档中出现了500次，world出现了100次
```

3.Field-length(字段长度归约) norm:field越长，相关度越低

搜索内容：hello world

```
{"title":"hello,what's your name?","content":{"owieurowieuolsdjflk"}}

{"title":"hi,good morning","content":{"lkjkljkj.......world"}}
```

查看分数是如何计算的：

```
GET /lib3/user/_search?explain=true
{
    "query":{
        "match":{
            "interests": "duanlian,changge"
        }
    }
}
```

查看一个文档能否匹配上某个查询：

```
GET /lib3/user/2/_explain
{
    "query":{
        "match":{
            "interests": "duanlian,changge"
        }
    }
}
```

3.24 Doc Values 解析

DocValues其实是Lucene在构建倒排索引时，会额外建立一个有序的正排索引(基于document => field value的映射列表)

```
{"birthday":"1985-11-11",age:23}

{"birthday":"1989-11-11",age:29}

document     age       birthday

doc1         23         1985-11-11

doc2         29         1989-11-11
```

存储在磁盘上，节省内存

对排序，分组和一些聚合操作能够大大提升性能

注意：默认对不分词的字段是开启的，对分词字段无效（需要把fielddata设置为true）

```
PUT /lib3
{
    "settings":{
    "number_of_shards" : 3,
```

```
    "number_of_replicas" : 0
    },
     "mappings":{
      "user":{
        "properties":{
            "name": {"type":"text"},
            "address": {"type":"text"},
            "age": {
              "type":"integer",
              "doc_values":false
            },
            "interests": {"type":"text"},
            "birthday": {"type":"date"}
        }
      }
     }
}
```

3.25 基于scroll技术滚动搜索大量数据

如果一次性要查出来比如10万条数据，那么性能会很差，此时一般会采取用scoll滚动查询，一批一批的查，直到所有数据都查询完为止。

1.scoll搜索会在第一次搜索的时候，保存一个当时的视图快照，之后只会基于该旧的视图快照提供数据搜索，如果这个期间数据变更，是不会让用户看到的

2.采用基于*doc(不使用*score)进行排序的方式，性能较高

3.每次发送scroll请求，我们还需要指定一个scoll参数，指定一个时间窗口，每次搜索请求只要在这个时间窗口内能完成就可以了

```
GET /lib3/user/_search?scroll=1m
{
  "query": {
    "match_all": {}
  },
  "sort":["_doc"],
  "size":3
}

GET /_search/scroll
{
   "scroll": "1m",
   "scroll_id":
"DnF1ZXJ5VGhlbkZldGNoAwAAAAAAAAAdFkEwRENOVTdnUUJPWVZUd1p2WE5hV2cAAAAAAAAAAHhZBM
ERDTlU3Z1FCT1lWVHdadlhOYVdnAAAAAAAAAAB8WQTBEQ05VN2dRQk9ZVlR3WnZYTmFXZw=="
}
```

3.26 dynamic mapping策略

dynamic:

1.true:遇到陌生字段就 dynamic mapping

2.false:遇到陌生字段就忽略

3.strict:约到陌生字段就报错

```
PUT /lib8
{
    "settings":{
    "number_of_shards" : 3,
    "number_of_replicas" : 0
    },
     "mappings":{
      "user":{
        "dynamic":strict,
        "properties":{
            "name": {"type":"text"},
            "address":{
                "type":"object",
                "dynamic":true
            },
        }
      }
     }
}

#会报错

PUT  /lib8/user/1
{
  "name":"lisi",
  "age":20,
  "address":{
    "province":"beijing",
    "city":"beijing"
  }
}
```

date_detection:默认会按照一定格式识别date，比如yyyy-MM-dd

可以手动关闭某个type的date_detection

```
PUT /lib8
{
    "settings":{
    "number_of_shards" : 3,
    "number_of_replicas" : 0
    },
     "mappings":{
      "user":{
        "date_detection": false,
        }
    }
}
```

定制 dynamic mapping template(type)

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "dynamic_templates": [
        {
          "en": {
            "match": "*_en",
            "match_mapping_type": "string",
            "mapping": {
              "type": "text",
              "analyzer": "english"
            }
          }
        }
      ]
     }
  }
}
#使用了模板

PUT /my_index/my_type/3
{
  "title_en": "this is my dog"

}
#没有使用模板

PUT /my_index/my_type/5
{
  "title": "this is my cat"
}
```

```
GET my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "is"
    }
  }
}
```

3.27 重建索引

一个field的设置是不能修改的，如果要修改一个field，那么应该重新按照新的mapping，建立一个index，然后将数据批量查询出来，重新用bulk api写入到index中。

批量查询的时候，建议采用scroll api，并且采用多线程并发的方式来reindex数据，每次scroll就查询指定日期的一段数据，交给一个线程即可。

```
PUT /index1/type1/4
{
   "content":"1990-12-12"
}

GET /index1/type1/_search

GET /index1/type1/_mapping



#报错
PUT /index1/type1/4
{
   "content":"I am very happy."
}
```

修改content的类型为string类型,报错，不允许修改

```
PUT /index1/_mapping/type1
{
  "properties": {
    "content":{
      "type": "text"
    }
  }
}
```

创建一个新的索引，把index1索引中的数据查询出来导入到新的索引中 #但是应用程序使用的是之前的索引，为了不用重启应用程序，给index1这个索引起个#别名

```
PUT /index1/_alias/index2
```

#创建新的索引，把content的类型改为字符串

```
PUT /newindex
{
  "mappings": {
    "type1":{
      "properties": {
        "content":{
          "type": "text"
        }
      }
    }
  }
}
```

#使用scroll批量查询

```
GET /index1/type1/_search?scroll=1m
{
  "query": {
    "match_all": {}
  },
  "sort": ["_doc"],
  "size": 2
}
```

#使用bulk批量写入新的索引

```
POST /_bulk
{"index":{"_index":"newindex","_type":"type1","_id":1}}
{"content":"1982-12-12"}
```

#将别名index2和新的索引关联，应用程序不用重启

```
POST /_aliases
{
  "actions": [
    {"remove": {"index":"index1","alias":"index2"}},
    {"add": {"index": "newindex","alias": "index2"}}
]
}

GET index2/type1/_search
```

3.28 索引不可变的原因

倒排索引包括：

文档的列表，文档的数量，词条在每个文档中出现的次数，出现的位置，每个文档的长度，所有文档的平均长度

索引不变的原因：

不需要锁，提升了并发性能

可以一直保存在缓存中（filter）

节省cpu和io开销