ElasticSearch

版本：V 1.2.2

# 一 elasticsearch简介

ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口。Elasticsearch是用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。

## 1 elasticSearch的使用场景

1. 为用户提供按关键字查询的全文搜索功能。
2. 实现企业海量数据的处理分析的解决方案。大数据领域的重要一份子，如著名的ELK框架(ElasticSearch,Logstash,Kibana)。

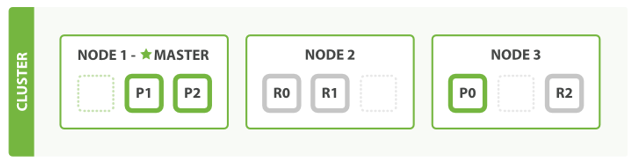
## 2 与其他数据存储进行比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | redis | mysql | elasticsearch | hbase | hadoop/hive |
| 容量/容量扩展 | 低 | 中 | 较大 | 海量 | 海量 |
| 查询时效性 | 极高 | 中等 | 较高 | 中等 | 低 |
| 查询灵活性 | 较差 k-v模式 | 非常好，支持sql | 较好，关联查询较弱，但是可以全文检索，DSL语言可以处理过滤、匹配、排序、聚合等各种操作 | 较差，主要靠rowkey,  scan的话性能不行，或者建立二级索引 | 非常好，支持sql |
| 写入速度 | 极快 | 中等 | 较快 | 较快 | 慢 |
| 一致性、事务 | 弱 | 强 | 弱 | 弱 | 弱 |

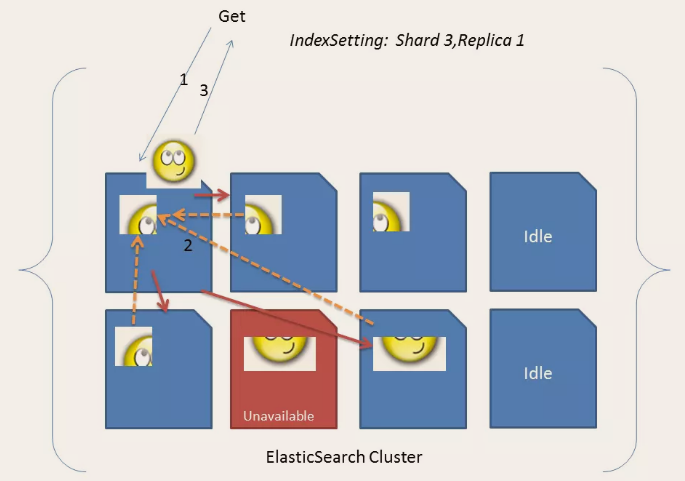
## 3 elasticsearch的特点

### 3.1 天然分片，天然集群

es 把数据分成多个shard，下图中的P0-P2，多个shard可以组成一份完整的数据，这些shard可以分布在集群中的各个机器节点中。随着数据的不断增加，集群可以增加多个分片，把多个分片放到多个机子上，已达到负载均衡，横向扩展。



在实际运算过程中，每个查询任务提交到某一个节点，该节点必须负责将数据进行整理汇聚，再返回给客户端，也就是一个简单的节点上进行Map计算，在一个固定的节点上进行Reduces得到最终结果向客户端返回。



这种集群分片的机制造就了elasticsearch强大的数据容量及运算扩展性。

### 3.2 天然索引

ES 所有数据都是默认进行索引的，这点和mysql正好相反，mysql是默认不加索引，要加索引必须特别说明，ES只有不加索引才需要说明。

而ES使用的是倒排索引和Mysql的B+Tree索引不同。

#### 传统关系性数据库



弊端：

1、 对于传统的关系性数据库对于关键词的查询，只能逐字逐行的匹配，性能非常差。

2、匹配方式不合理，比如搜索“小密手机” ，如果用like进行匹配， 根本匹配不到。但是考虑使用者的用户体验的话，除了完全匹配的记录，还应该显示一部分近似匹配的记录，至少应该匹配到“手机”。

#### 倒排索引是怎么处理的

全文搜索引擎目前主流的索引技术就是倒排索引的方式。

 传统的保存数据的方式都是

记录→单词

而倒排索引的保存数据的方式是

单词→记录

例如

搜索“红海行动”

但是数据库中保存的数据如图：

那么搜索引擎是如何能将两者匹配上的呢？

基于分词技术构建倒排索引：

首先每个记录保存数据时，都不会直接存入数据库。系统先会对数据进行分词，然后以倒排索引结构保存。如下：



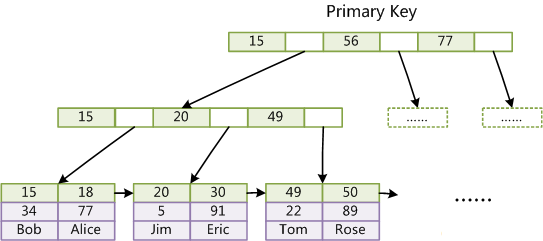
然后等到用户搜索的时候，会把搜索的关键词也进行分词，会把“红海行动”分词分成：红海和行动两个词。

这样的话，先用红海进行匹配，得到id=1和id=2的记录编号，再用行动匹配可以迅速定位id为1,3的记录。

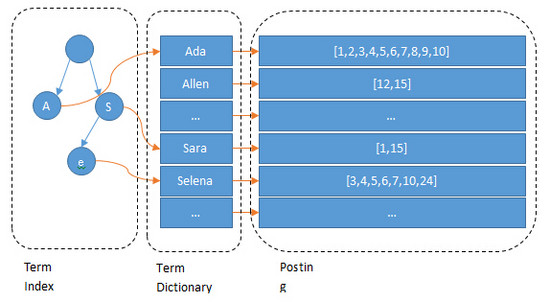
那么全文索引通常，还会根据匹配程度进行打分，显然1号记录能匹配的次数更多。所以显示的时候以评分进行排序的话，1号记录会排到最前面。而2、3号记录也可以匹配到。

#### 索引结构对比

##### B+Tree



##### lucene 倒排索引结构



可以看到 lucene 为倒排索引(Term Dictionary)部分又增加一层Term Index结构，用于快速定位，而这Term Index是缓存在内存中的，但mysql的B+tree不在内存中，所以整体来看ES速度更快，但同时也更消耗资源（内存、磁盘）。

## 4 lucene与elasticsearch的关系

咱们之前讲的处理分词，构建倒排索引，等等，都是这个叫lucene的做的。那么能不能说这个lucene就是搜索引擎呢？

还不能。lucene只是一个提供全文搜索功能类库的核心工具包，而真正使用它还需要一个完善的服务框架搭建起来的应用。

好比lucene是类似于发动机，而搜索引擎软件（ES,Solr）就是汽车。

目前市面上流行的搜索引擎软件，主流的就两款，elasticsearch和solr,这两款都是基于lucene的搭建的，可以独立部署启动的搜索引擎服务软件。由于内核相同，所以两者除了服务器安装、部署、管理、集群以外，对于数据的操作，修改、添加、保存、查询等等都十分类似。就好像都是支持sql语言的两种数据库软件。只要学会其中一个另一个很容易上手。

从实际企业使用情况来看，elasticSearch的市场份额逐步在取代solr，国内百度、京东、新浪都是基于elasticSearch实现的搜索功能。国外就更多了 像维基百科、GitHub、Stack Overflow等等也都是基于ES的。

# 二 elasticSearch的安装

**详见《elasticSearch的安装手册》**

# 三 elasticsearch的基本概念

|  |  |
| --- | --- |
| cluster | 整个elasticsearch 默认就是集群状态，整个集群是一份完整、互备的数据。 |
| node | 集群中的一个节点，一般只一个进程就是一个node |
| shard | 分片，即使是一个节点中的数据也会通过hash算法，分成多个片存放，默认是5片。（7.0默认改为1片） |
| index | 相当于rdbms的database(5.x), 对于用户来说是一个逻辑数据库，虽然物理上会被分多个shard存放，也可能存放在多个node中。 6.x 7.x index相当于table |
| type | 类似于rdbms的table，但是与其说像table，其实更像面向对象中的class , 同一Json的格式的数据集合。（6.x只允许建一个，7.0被废弃，造成index实际相当于table级） |
| document | 类似于rdbms的 row、面向对象里的object |
| field | 相当于字段、属性 |

GET /\_cat/nodes?v 查询各个节点状态

GET /\_cat/indices?v 查询各个索引状态

GET /\_cat/shards/xxxx 查询某个索引的分片情况

# 四 elasticsearch restful api (DSL)

DSL全称 Domain Specific language，即特定领域专用语言。

### 1、es中保存的数据结构

|  |
| --- |
| public class Movie {  String id;  String name;  Double doubanScore;  List<Actor> actorList;  }  public class Actor{  String id;  String name;  } |

这两个对象如果放在关系型数据库保存，会被拆成2张表，但是elasticsearch是用一个json来表示一个document。

所以他保存到es中应该是：

|  |
| --- |
| {  "id":"1",  "name":"operation red sea",  "doubanScore":"8.5",  "actorList":[  {"id":"1","name":"zhangyi"},  {"id":"2","name":"haiqing"},  {"id":"3","name":"zhanghanyu"}  ]  } |

### 2 对数据的操作

#### 2.1 查看es中有哪些索引

|  |
| --- |
| GET /\_cat/indices?v |

es 中会默认存在一个名为.kibana的索引

表头的含义

|  |  |
| --- | --- |
| health | green(集群完整) yellow(单点正常、集群不完整) red(单点不正常) |
| status | 是否能使用 |
| index | 索引名 |
| uuid | 索引统一编号 |
| pri | 主节点几个 |
| rep | 从节点几个 |
| docs.count | 文档数 |
| docs.deleted | 文档被删了多少 |
| store.size | 整体占空间大小 |
| pri.store.size | 主节点占 |

#### 2.2 增加一个索引

|  |
| --- |
| PUT /movie\_index |

#### 2.3 删除一个索引

ES 是不删除也不修改任何数据的，而是增加版本号

|  |
| --- |
| DELETE /movie\_index |

#### 2.4 新增文档

1. 格式 PUT /index/type/id

|  |
| --- |
| PUT /movie\_index/movie/1  { "id":1,  "name":"operation red sea",  "doubanScore":8.5,  "actorList":[  {"id":1,"name":"zhang yi"},  {"id":2,"name":"hai qing"},  {"id":3,"name":"zhang han yu"}  ]  }  PUT /movie\_index/movie/2  {  "id":2,  "name":"operation meigong river",  "doubanScore":8.0,  "actorList":[  {"id":3,"name":"zhang han yu"}  ]  }  PUT /movie\_index/movie/3  {  "id":3,  "name":"incident red sea",  "doubanScore":5.0,  "actorList":[  {"id":4,"name":"zhang chen"}  ]  } |

如果之前没建过index或者type，es 会自动创建。

#### 2.5 直接用id查找

|  |
| --- |
| GET movie\_index/movie/1 |

#### 2.6 修改—整体替换

和新增没有区别 要求：必须包括全部字段

|  |
| --- |
| PUT /movie\_index/movie/3  {  "id":"3",  "name":"incident red sea",  "doubanScore":"5.0",  "actorList":[  {"id":"1","name":"zhang chen"}  ]  } |

#### 2.7修改—某个字段

|  |
| --- |
| POST movie\_index/movie/3/\_update  {  "doc": {  "doubanScore":"7.0"  }  } |

#### 2.8 删除一个document

|  |
| --- |
| DELETE movie\_index/movie/3 |

#### 2.9 搜索type全部数据

|  |
| --- |
| GET movie\_index/movie/\_search |

结果

|  |
| --- |
| {  "took": 2, //耗费时间 毫秒  "timed\_out": false, //是否超时  "\_shards": {  "total": 5, //发送给全部5个分片  "successful": 5,  "skipped": 0,  "failed": 0  },  "hits": {  "total": 3, //命中3条数据  "max\_score": 1, //最大评分  "hits": [ // 结果  {  "\_index": "movie\_index",  "\_type": "movie",  "\_id": 2,  "\_score": 1,  "\_source": {  "id": "2",  "name": "operation meigong river",  "doubanScore": 8.0,  "actorList": [  {  "id": "1",  "name": "zhang han yu"  }  ]  }  。。。。。。。。  。。。。。。。。  } |

#### 2.10 按条件查询(全部)

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match\_all": {}  }  } |

#### 2.11 按分词查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red"}  }  } |

#### 2.12 按分词子属性查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"actorList.name":"zhang"}  }  } |

#### 2.13 match phrase

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match\_phrase": {"name":"operation red"}  }  } |

按短语查询，不再利用分词技术，直接用短语在原始数据中匹配

#### 2.14 fuzzy查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "fuzzy": {"name":"rad"}  }  } |

校正匹配分词，当一个单词都无法准确匹配，es通过一种算法对非常接近的单词也给与一定的评分，能够查询出来，但是消耗更多的性能。

#### 2.15 过滤--查询后过滤

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red"}  },  "post\_filter":{  "term": {  "actorList.id": 3  }  }  } |

#### 2.16 过滤--查询前过滤（推荐使用）

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "bool":{  "filter":[ {"term": { "actorList.id": "1" }},  {"term": { "actorList.id": "3" }}  ],  "must":{"match":{"name":"red"}}  }  }  } |

#### 2.17 过滤--按范围过滤

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query": {  "bool": {  "filter": {  "range": {  "doubanScore": {"gte": 8}  }  }  }  }  } |

关于范围操作符：

|  |  |
| --- | --- |
| gt | 大于 |
| lt | 小于 |
| gte | 大于等于 great than or equals |
| lte | 小于等于 less than or equals |

#### 2.18 排序

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red sea"}  }  , "sort": [  {  "doubanScore": {  "order": "desc"  }  }  ]  } |

#### 2.19 分页查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query": { "match\_all": {} },  "from": 1,  "size": 1  } |

#### 2.20 指定查询的字段

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query": { "match\_all": {} },  "\_source": ["name", "doubanScore"]  } |

#### 2.21 高亮

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red sea"}  },  "highlight": {  "fields": {"name":{} }  }    } |

#### 2.22 聚合

取出每个演员共参演了多少部电影

|  |
| --- |
| GET movie\_index/movie/\_search  {  "aggs": {  "groupby\_actor": {  "terms": {  "field": "actorList.name.keyword"  }  }  }  } |

每个演员参演电影的平均分是多少，并按评分排序

|  |
| --- |
| GET movie\_index/movie/\_search  {  "aggs": {  "groupby\_actor\_id": {  "terms": {  "field": "actorList.name.keyword" ,  "order": {  "avg\_score": "desc"  }  },  "aggs": {  "avg\_score":{  "avg": {  "field": "doubanScore"  }  }  }  }  }  } |

聚合时为何要加 .keyword后缀？

.keyword 是某个字符串字段，专门储存不分词格式的副本 ，在某些场景中只允许只用不分词的格式，比如过滤filter 比如 聚合aggs, 所以字段要加上.keyword的后缀。

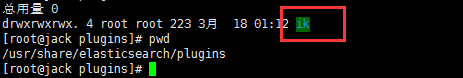
### 3 中文分词

elasticsearch本身自带的中文分词，就是单纯把中文一个字一个字的分开，根本没有词汇的概念。但是实际应用中，用户都是以词汇为条件，进行查询匹配的，如果能够把文章以词汇为单位切分开，那么与用户的查询条件能够更贴切的匹配上，查询速度也更加快速。

分词器下载网址：<https://github.com/medcl/elasticsearch-analysis-ik>

#### 3.1 安装

下载好的zip包，请解压后放到 …/elasticsearch/plugins/ik



然后重启es

#### 3.2测试使用

使用默认

|  |
| --- |
| GET movie\_index/\_analyze  {  "text": "我是中国人"  } |

请观察结果

使用分词器

|  |
| --- |
| GET movie\_index/\_analyze  { "analyzer": "ik\_smart",  "text": "我是中国人"  } |

请观察结果

另外一个分词器

ik\_max\_word

|  |
| --- |
| GET movie\_index/\_analyze  { "analyzer": "ik\_max\_word",  "text": "我是中国人"  } |

请观察结果

能够看出不同的分词器，分词有明显的区别，所以以后定义一个type不能再使用默认的mapping了，要手工建立mapping, 因为要选择分词器。

#### 3.3 自定义词库

修改/usr/share/elasticsearch/plugins/ik/config/中的IKAnalyzer.cfg.xml

|  |
| --- |
| <?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"></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">http://192.168.67.163/fenci/myword.txt</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> |

按照标红的路径利用nginx发布静态资源

在nginx.conf中配置

|  |
| --- |
| server {  listen 80;  server\_name 192.168.67.163;  location /fenci/ {  root es;  }  } |

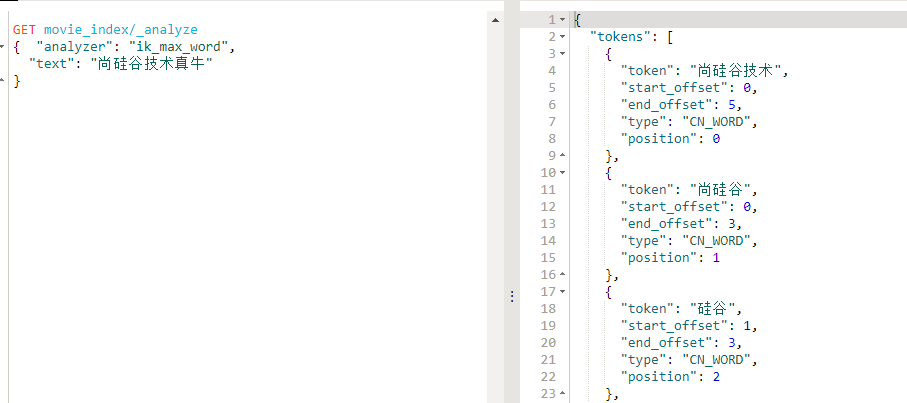
并且在/usr/local/nginx/下建/es/fenci/目录，目录下加myword.txt

myword.txt中编写关键词，每一行代表一个词。



然后重启es服务器，重启nginx。

在kibana中测试分词效果



更新完成后，es只会对新增的数据用新词分词。历史数据是不会重新分词的。如果想要历史数据重新分词。需要执行：

|  |
| --- |
| POST movies\_index\_chn/\_update\_by\_query?conflicts=proceed |

### 4 关于mapping

之前说type可以理解为table，那每个字段的数据类型是如何定义的呢

#### 4.1 查看mapping

|  |
| --- |
| GET movie\_index/\_mapping/movie |

实际上每个type中的字段是什么数据类型，由mapping定义。

但是如果没有设定mapping系统会自动，根据一条数据的格式来推断出应该的数据格式。

* true/false → boolean
* 1020 → long
* 20.1 → double
* “2018-02-01” → date
* “hello world” → text +keyword

默认只有text会进行分词，keyword是不会分词的字符串。

mapping除了自动定义，还可以手动定义，但是只能对新加的、没有数据的字段进行定义。一旦有了数据就无法再做修改了。

注意：虽然每个Field的数据放在不同的type下,但是同一个名字的Field在一个index下只能有一种mapping定义。

#### 4.2基于中文分词搭建索引

1、建立mapping

|  |
| --- |
| PUT movie\_chn  {  "mappings": {  "movie":{  "properties": {  "id":{  "type": "long"  },  "name":{  "type": "text"  , "analyzer": "ik\_smart"  },  "doubanScore":{  "type": "double"  },  "actorList":{  "properties": {  "id":{  "type":"long"  },  "name":{  "type":"keyword"  }  }  }  }  }  }  } |

插入数据

|  |
| --- |
| PUT /movie\_chn/movie/1  { "id":1,  "name":"红海行动",  "doubanScore":8.5,  "actorList":[  {"id":1,"name":"张译"},  {"id":2,"name":"海清"},  {"id":3,"name":"张涵予"}  ]  }  PUT /movie\_chn/movie/2  {  "id":2,  "name":"湄公河行动",  "doubanScore":8.0,  "actorList":[  {"id":3,"name":"张涵予"}  ]  }  PUT /movie\_chn/movie/3  {  "id":3,  "name":"红海事件",  "doubanScore":5.0,  "actorList":[  {"id":4,"name":"张晨"}  ]  } |

查询测试

|  |
| --- |
| GET /movie\_chn/movie/\_search  {  "query": {  "match": {  "name": "红海战役"  }  }  }  GET /movie\_chn/movie/\_search  {  "query": {  "term": {  "actorList.name": "张译"  }  }  } |

### 5索引别名 \_aliases

**索引别名**就像一个快捷方式或软连接，可以指向一个或多个索引，也可以给任何一个需要索引名的API来使用。别名 带给我们极大的灵活性，允许我们做下面这些：

1. 给多个索引分组 (例如， last\_three\_months)
2. 给索引的一个子集创建视图
3. 在运行的集群中可以无缝的从一个索引切换到另一个索引

#### 5.1 创建索引别名

建表时直接声明

|  |
| --- |
| PUT movie\_chn\_2020  { "aliases": {  "movie\_chn\_2020-query": {}  },  "mappings": {  "movie":{  "properties": {  "id":{  "type": "long"  },  "name":{  "type": "text"  , "analyzer": "ik\_smart"  },  "doubanScore":{  "type": "double"  },  "actorList":{  "properties": {  "id":{  "type":"long"  },  "name":{  "type":"keyword"  }  }  }  }  }  }  } |

为已存在的索引增加别名

|  |
| --- |
| POST \_aliases  {  "actions": [  { "add": { "index": "movie\_chn\_xxxx", "alias": "movie\_chn\_2020-query" }}  ]  } |

也可以通过加过滤条件缩小查询范围，建立一个子集视图

|  |
| --- |
| POST \_aliases  {  "actions": [  { "add":  { "index": "movie\_chn\_xxxx",  "alias": "movie\_chn0919-query-zhhy",  "filter": {  "term": { "actorList.id": "3"  }  }  }  }  ]  } |

#### 5.2 查询别名。 与使用普通索引没有区别

|  |
| --- |
| GET movie\_chn\_2020-query/\_search |

#### 5.3 删除某个索引的别名

|  |
| --- |
| POST \_aliases  {  "actions": [  { "remove": { "index": "movie\_chn\_xxxx", "alias": "movie\_chn\_2020-query" }}  ]  } |

#### 5.4 为某个别名进行无缝切换

|  |
| --- |
| POST /\_aliases  {  "actions": [  { "remove": { "index": "movie\_chn\_xxxx", "alias": "movie\_chn\_2020-query" }},  { "add": { "index": "movie\_chn\_yyyy", "alias": "movie\_chn\_2020-query" }}  ]  } |

#### 5.5查询别名列表

|  |
| --- |
| GET \_cat/aliases?v |

### 6 索引模板

Index Template 索引模板，顾名思义，就是创建索引的模具，其中可以定义一系列规则来帮助我们构建符合特定业务需求的索引的 mappings 和 settings，通过使用 Index Template 可以让我们的索引具备可预知的一致性。

#### 6.1 常见的场景: 分割索引

**分割索引**就是根据时间间隔把一个业务索引切分成多个索引。

比如 把order\_info 变成 order\_info\_20200101,order\_info\_20200102 …..

这样做的好处有两个：

1. **结构变化的灵活性**：因为elasticsearch不允许对数据结构进行修改。但是实际使用中索引的结构和配置难免变化，那么只要对下一个间隔的索引进行修改，原来的索引位置原状。这样就有了一定的灵活性。
2. **查询范围优化**： 因为一般情况并不会查询全部时间周期的数据，那么通过切分索引，物理上减少了扫描数据的范围，也是对性能的优化。

#### 6.2 创建模板

|  |
| --- |
| PUT \_template/template\_movie2020  {  "index\_patterns": ["movie\_test\*"],  "settings": {  "number\_of\_shards": 1  },  "aliases" : {  "{index}-query": {},  "movie\_test-query":{}  },  "mappings": {  "\_doc": {  "properties": {  "id": {  "type": "keyword"  },  "movie\_name": {  "type": "text",  "analyzer": "ik\_smart"  }  }  }  }  } |

其中 "index\_patterns": ["movie\_test\*"], 的含义就是凡是往movie\_test开头的索引写入数据时，如果索引不存在，那么es会根据此模板自动建立索引。

在 "aliases" 中用{index}表示，获得真正的创建的索引名。

测试

|  |
| --- |
| POST movie\_test\_2020xxxx/\_doc  {  "id":"333",  "name":"zhang3"  } |

#### 6.3 查看系统中已有的模板清单

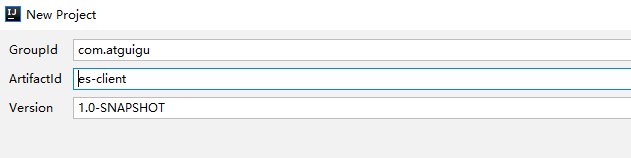
|  |
| --- |
| GET \_cat/templates |

#### 6.4查看某个模板详情

|  |
| --- |
| GET \_template/template\_movie2020  或者  GET \_template/template\_movie\* |

# 五 Java程序中的应用

### 1 、搭建模块



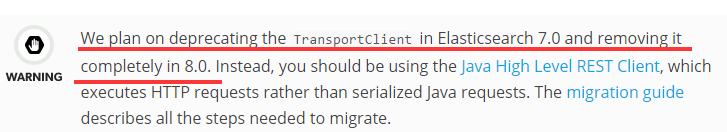
### 2、 关于es 的java 客户端的选择

目前市面上有两类客户端

一类是TransportClient 为代表的ES原生客户端，不能执行原生dsl语句必须使用它的Java api方法。

另外一种是以Rest Api为主的missing client，最典型的就是jest。 这种客户端可以直接使用dsl语句拼成的字符串，直接传给服务端，然后返回json字符串再解析。

两种方式各有优劣，但是最近elasticsearch官网，宣布计划在7.0以后的版本中废除TransportClient。以RestClient为主。



所以在官方的RestClient 基础上，进行了简单包装的Jest客户端，就成了首选，而且该客户端也与springboot完美集成。

### 3 、导入Jest依赖

|  |
| --- |
| <**dependency**>  <**groupId**>io.searchbox</**groupId**>  <**artifactId**>jest</**artifactId**>  <**version**>5.3.3</**version**>  </**dependency**>  <**dependency**>  <**groupId**>net.java.dev.jna</**groupId**>  <**artifactId**>jna</**artifactId**>  <**version**>4.5.2</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.codehaus.janino</**groupId**>  <**artifactId**>commons-compiler</**artifactId**>  <**version**>2.7.8</**version**> </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.elasticsearch/elasticsearch -->* <**dependency**>  <**groupId**>org.elasticsearch</**groupId**>  <**artifactId**>elasticsearch</**artifactId**>  <**version**>2.4.6</**version**> </**dependency**> |

### 4 、在测试类中测试ES

测试类

|  |
| --- |
| **object** EsClient {   **private var** *factory*: JestClientFactory=**null**;   **def** getClient:JestClient ={  **if**(*factory*==**null**)*build*();  *factory*.getObject   }   **def** build(): Unit ={  *factory*=**new** JestClientFactory  *factory*.setHttpClientConfig(**new** HttpClientConfig.Builder(**"http://hadoop1:9200"** )  .multiThreaded(**true**)  .maxTotalConnection(20)  .connTimeout(10000).readTimeout(1000).build())   }      **def** putIndex(args: Array[String]): Unit = {   **val** jest: JestClient = *getClient* **val** actorList=**new** util.ArrayList[String]()  actorList.add(**"tom"**)  actorList.add(**"Jack"**)  **val** movieTest = MovieTest(**"102"**,**"中途岛之战"**,actorList)  **val** datestring: String = **new** SimpleDateFormat(**"yyyyMMdd"**).format(**new** Date)  **val** index: Index = **new** Index.Builder(movieTest).index(**"movie\_test1021\_"**+datestring).`type`(**"\_doc"**).id(movieTest.id).build()   jest.execute(index)  jest.close()   }   **def** main(args: Array[String]): Unit = {  **val** jest: JestClient = *getClient* **var** query=**"{\n \"query\": {\n \"bool\": {\n \"should\": [\n {\"match\": {\n \"name\": \"red sea\"\n }}\n ],\n \"filter\": {\n \"range\": {\n \"doubanScore\": {\n \"gte\": 3\n }\n }\n }\n } \n \n }\n ,\n \"sort\": [\n {\n \"doubanScore\": {\n \"order\": \"asc\"\n }\n }\n ],\n \"from\": 0, \n \"size\": 20, \n \"highlight\": {\n \"fields\": { \n \"name\": {}\n }\n } \n}"**;   **val** searchBuilder = **new** SearchSourceBuilder  **val** boolbuilder = **new** BoolQueryBuilder()  boolbuilder.should(**new** MatchQueryBuilder(**"name"**,**"red sea"**))  .filter(**new** RangeQueryBuilder(**"doubanScore"**).gte(3))  searchBuilder.query(boolbuilder);  searchBuilder.sort(**"doubanScore"**,SortOrder.*ASC*)  searchBuilder.from(0)  searchBuilder.size(20)  searchBuilder.highlight(**new** HighlightBuilder().field(**"name"**))    **val** query2: String = searchBuilder.toString  *println*(query2)  **val** search: Search = **new** Search.Builder(query2).addIndex(**"movie1021\_index"**).addType(**"movie"**).build()  **val** result: SearchResult = jest.execute(search)  **val** hitList: util.List[SearchResult#Hit[util.Map[String, Any], Void]] = result.getHits(*classOf*[util.Map[String ,Any]])  **var** resultList:ListBuffer[util.Map[String,Any]]=**new** ListBuffer[util.Map[String, Any]] ;  **import** scala.collection.JavaConversions.\_  **for** ( hit <- hitList ) {  **val** source: util.Map[String, Any] = hit.*source* resultList.add(source)  }  *println*(resultList.mkString(**"\n"**))   jest.close()  }    **case class** MovieTest(id:String ,movie\_name:String, actionNameList: java.util.List[String] ){   }    } |