一、ElasticSearch文档分值_score计算底层原理

1）boolean model

根据用户的query条件，先过滤出包含指定term的doc

```
query "hello world" --> hello / world / hello & world

bool --> must/must not/should --> 过滤 --> 包含 / 不包含 / 可能包含

doc --> 不打分数 --> 正或反 true or false --> 为了减少后续要计算的doc的数量，提升性能
```

2)relevance score算法，简单来说，就是计算出，一个索引中的文本，与搜索文本，他们之间的关联匹配程度

Elasticsearch使用的是 term frequency/inverse document frequency算法，简称为TF/IDF算法

Term frequency：搜索文本中的各个词条在field文本中出现了多少次，出现次数越多，就越相关

```
搜索请求：hello world


doc1：hello you，and world is very good

doc2：hello，how are you

```

Inverse document frequency：搜索文本中的各个词条在整个索引的所有文档中出现了多少次，出现的次数越多，就越不相关

```
搜索请求：hello world


doc1：hello，tuling is very good
```

```
6
7 doc2: hi world, how are you
8
```

比如说，在index中有1万条document，hello这个单词在所有的document中，一共出现了1000次；world这个单词在所有的document中，一共出现了100次

Field-length norm：field长度，field越长，相关度越弱

搜索请求：hello world

```
1 doc1: { "title": "hello article", "content": "...... N个单词" }
2
3 doc2: { "title": "my article", "content": "...... N个单词，hi world" }
4
```

hello world在整个index中出现的次数是一样多的

doc1更相关，title field更短

2、分析一个document上的_score是如何被计算出来的

```
GET /es_db/_doc/1/_explain
{
 "query": {
 "match": {
 "remark": "java developer"
 }
 }
}
```

3、vector space model

多个term对一个doc的总分数

hello world --> es会根据hello world在所有doc中的评分情况，计算出一个query vector，query向量

hello这个term，给的基于所有doc的一个评分就是2
world这个term，给的基于所有doc的一个评分就是5

[2, 5]

query vector

doc vector，3个doc，一个包含1个term，一个包含另一个term，一个包含2个term

3个doc

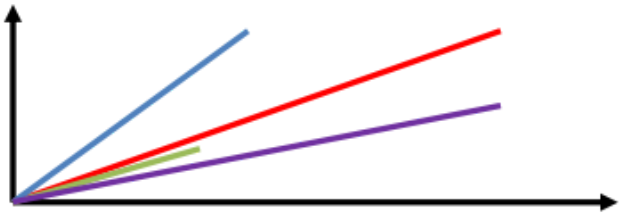
doc1：包含hello --> [2, 0]
doc2：包含world --> [0, 5]
doc3：包含hello, world --> [2, 5]

会给每一个doc，拿每个term计算出一个分数来，hello有一个分数，world有一个分数，再拿所有term的分数组成一个doc vector

画在一个图中，取每个doc vector对query vector的弧度，给出每个doc对多个term的总分数

每个doc vector计算出对query vector的弧度，最后基于这个弧度给出一个doc相对于query中多个term的总分数
弧度越大，分数月底; 弧度越小，分数越高

如果是多个term，那么就是线性代数来计算，无法用图表示



二、es生产集群部署之针对生产集群的脑裂问题专门定制的重要参数

集群脑裂是什么？

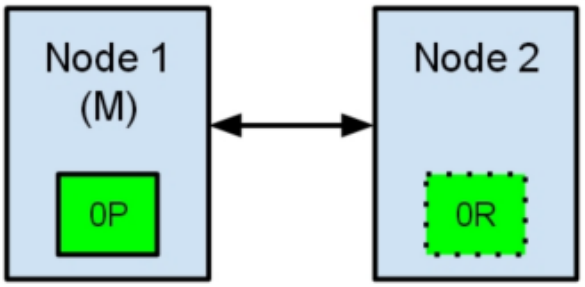
所谓脑裂问题，就是同一个集群中的不同节点，对于集群的状态有了不一样的理解，比如集群中存在两个master

如果因为网络的故障，导致一个集群被划分成了两片，每片都有多个node，以及一个master，那么集群中就出现了两个master了。

但是因为master是集群中非常重要的一个角色，主宰了集群状态的维护，以及shard的分配，

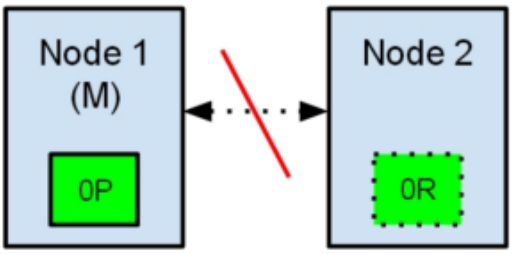
因此如果有两个master，可能会导致破坏数据。

如：



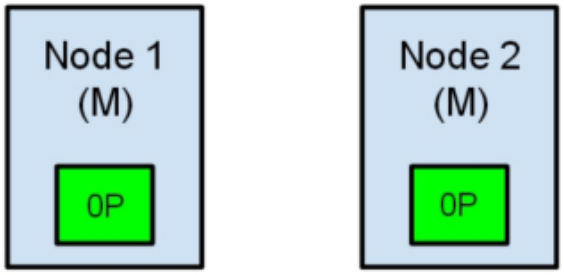
节点1在启动时被选举为主节点并保存主分片标记为0P，而节点2保存复制分片标记为0R

现在，如果在两个节点之间的通讯中断了，会发生什么？由于网络问题或只是因为其中一个节点无响应，这是有可能发生的。



两个节点都相信对方已经挂了。节点1不需要做什么，因为它本来就被选举为主节点。但是节点2会自动选举它自己为主节点，因为它相信集群的一部分没有主节点了。

在elasticsearch集群，是有主节点来决定将分片平均的分布到节点上的。节点2保存的是复制分片，但它相信主节点不可用了。所以它会自动提升复制节点为主节点。



现在我们的集群在一个不一致的状态了。打在节点1上的索引请求会将索引数据分配在主节点，同时打在节点2的请求会将索引数据放在分片上。在这种情况下，分片的两份数据分开了，如果不做一个全量的重索引很难对它们进行重排序。在更坏的情况下，一个对集群无感知的索引客户端（例如，使用REST接口的），这个问题非常透明难以发现，无论哪个节点被命中索引请求仍然在每次都会成功完成。问题只有在搜索数据时才会被隐约发现：取决于搜索请求命中了哪个节点，结果都会不同。

那么那个参数的作用，就是告诉es直到有足够的master候选节点时，才可以选举出一个master，否则就不要选举出一个master。这个参数必须被设置为集群中master候选节点的quorum数量，也就是大多数。至于quorum的算法，就是：master候选节点数量 / 2 + 1。

比如我们有10个节点，都能维护数据，也可以是master候选节点，那么quorum就是10 / 2 + 1 = 6。

如果我们有三个master候选节点，还有100个数据节点，那么quorum就是3 / 2 + 1 = 2

如果我们有2个节点，都可以是master候选节点，那么quorum是2 / 2 + 1 = 2。此时就有问题了，因为如果一个node挂掉了，那么剩下一个master候选节点，是无法满足quorum数量的，也就无法选举出新的master，集群就彻底挂掉了。此时就只能将这个参数设置为1，但是这就无法阻止脑裂的发生了。

2个节点，discovery.zen.minimum_master_nodes分别设置成2和1会怎么样

综上所述，一个生产环境的es集群，至少要有3个节点，同时将这个参数设置为quorum，也就是2。discovery.zen.minimum_master_nodes设置为2，如何避免脑裂呢？

那么这个是参数是如何避免脑裂问题的产生的呢？比如我们有3个节点，quorum是2.现在网络故障，1个节点在一个网络区域，另外2个节点在另外一个网络区域，不同的网络区域内无法通信。这个时候有两种情况情况：

（1）如果master是单独的那个节点，另外2个节点是master候选节点，那么此时那个单独的master节点因为没有指定数量的候选master node在自己当前所在的集群内，因此就会取消当前master的角色，尝试重新选举，但是无法选举成功。然后另外一个网络区域内的node因为无法连接到master，就会发起重新选举，因为有两个master候选节点，满足了quorum，因此可以成功选举出一个master。此时集群中就会还是只有一个master。

（2）如果master和另外一个node在一个网络区域内，然后一个node单独在一个网络区域内。那么此时那个单独的node因为连接不上master，会尝试发起选举，但是因为master候选节点数量不到quorum，因此无法选举出master。而另外一个网络区域内，原先的那个master还会继续工作。这也可以保证集群内只有一个master节点。

综上所述，集群中master节点的数量至少3台，三台主节点通过在elasticsearch.yml中配置discovery.zen.minimum_master_nodes: 2，就可以避免脑裂问题的产生。

二、数据建模

1、案例:设计一个用户document数据类型，其中包含一个地址数据的数组，这种设计方式相对复杂，但是在管理数据时，更加的灵活。

```
PUT /user_index
{
"mappings": {
"properties": {
"login_name" : {
"type" : "keyword"
},
"age " : {
"type" : "short"
 },
 "address" : {
 "properties": {
 "province" : {
 "type" : "keyword"
 },
 "city" : {
 "type" : "keyword"
```

```
},
"street" : {
"type" : "keyword"
}
}
}
}
}
}
```

但是上述的数据建模有其明显的缺陷，就是针对地址数据做数据搜索的时候，经常会搜索出不必要的数据，如：在下述数据环境中，搜索一个province为北京，city为天津的用户。

```
PUT /user_index/_doc/1
{
"login_name" : "jack",
"age" : 25,
"address" : [
{
"province" : "北京",
"city" : "北京",
"street" : "枫林三路"
},
{
"province" : "天津",
"city" : "天津",
"street" : "华夏路"
}
]
}
PUT /user_index/_doc/2
{
"login_name" : "rose",
"age" : 21,
"address" : [
{
"province" : "河北",
"city" : "廊坊",
"street" : "燕郊经济开发区"
},
{
"province" : "天津",
"city" : "天津",
"street" : "华夏路"
```

```
}
]
}
```

执行的搜索应该如下：

```
GET /user_index/_search
{
"query": {
"bool": {
"must": [
{
"match": {
"address.province": "北京"
}
},
{
"match": {
"address.city": "天津"
}
}
]
}
}
}
```

但是得到的结果并不准确，这个时候就需要使用nested object来定义数据建模。

2、nested object

使用nested object作为地址数组的集体类型，可以解决上述问题，document模型如下：

```
PUT /user_index
{
"mappings": {
"properties": {
"login_name" : {
"type" : "keyword"
},
"age" : {
"type" : "short"
},
"address" : {
"type": "nested",
"properties": {
"province" : {
```

```
"type" : "keyword"
},
"city" : {
"type" : "keyword"
},
"street" : {
"type" : "keyword"
}
}
}
}
}
}
```

这个时候就需要使用nested对应的搜索语法来执行搜索了，语法如下：

```
GET /user_index/_search
{
"query": {
"bool": {
"must": [
{
"nested": {
"path": "address",
"query": {
"bool": {
"must": [
{
"match": {
"address.province": "北京"
}
},
{
"match": {
"address.city": "天津"
}
}
]
}
}
}
}
]
}
}
```

虽然语法变的复杂了，但是在数据的读写操作上都不会有错误发生，是推荐的设计方式。
其原因是：
普通的数组数据在ES中会被扁平化处理，处理方式如下：（如果字段需要分词，会将分词数据保存在对应的字段位置，当然应该是一个倒排索引，这里只是一个直观的案例）

```
{
  "login_name" : "jack",
  "address.province" : [ "北京", "天津" ],
  "address.city" : [ "北京", "天津" ]
  "address.street" : [ "西三旗东路", "古文化街" ]
}
```

那么nested object数据类型ES在保存的时候不会有扁平化处理，保存方式如下：所以在搜索的时候一定会有需要的搜索结果。

```
{
  "login_name" : "jack"
}
{
  "address.province" : "北京",
  "address.city" : "北京",
  "address.street" : "西三旗东路"
}
{
   "address.province" : "北京",
   "address.city" : "北京",
   "address.street" : "西三旗东路",
}
```

三、父子关系数据建模

nested object的建模，有个不好的地方，就是采取的是类似冗余数据的方式，将多个数据都放在一起了，维护成本就比较高
每次更新，需要重新索引整个对象（包括跟对象和嵌套对象）

ES 提供了类似关系型数据库中 Join 的实现。使用 Join 数据类型实现，可以通过 Parent / Child 的关系，从而分离两个对象
父文档和子文档是两个独立的文档
更新父文档无需重新索引整个子文档。子文档被新增，更改和删除也不会影响到父文档和其他子文档。

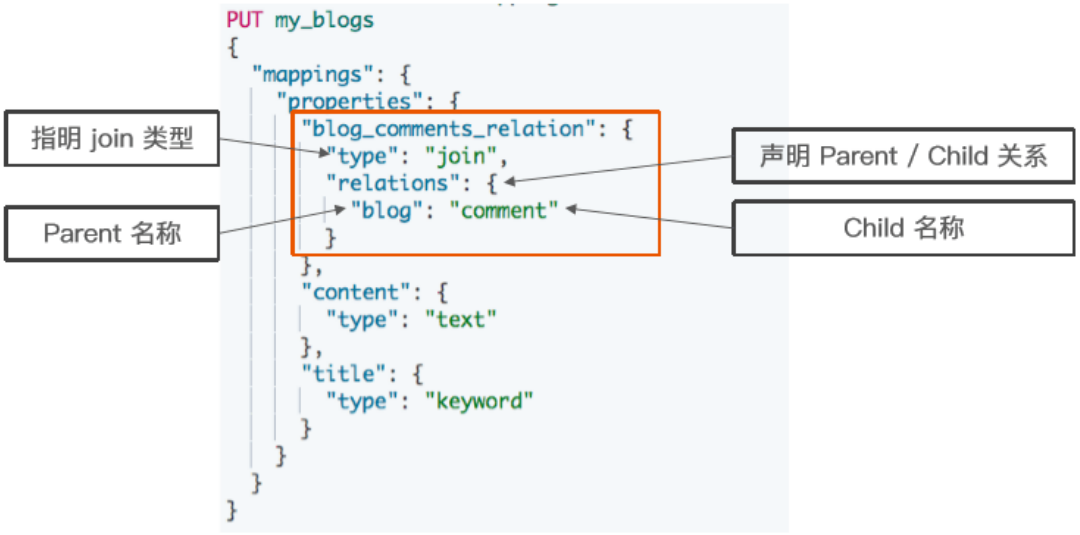
要点：父子关系元数据映射，用于确保查询时候的高性能，但是有一个限制，就是父子数据必须存在于一个shard中

父子关系数据存在一个shard中，而且还有映射其关联关系的元数据，那么搜索父子关系数据的时候，不用跨分片，一个分片本地自己就搞定了，性能当然高

父子关系

- 定义父子关系的几个步骤
  - 设置索引的 Mapping
  - 索引父文档
  - 索引子文档
  - 按需查询文档

设置 Mapping




```
DELETE my_blogs
# 设定 Parent/Child Mapping
PUT my_blogs
{

"mappings": {
"properties": {
"blog_comments_relation": {
"type": "join",
 "relations": {
 "blog": "comment"
```

```
}
},
"content": {
"type": "text"
},
"title": {
"type": "keyword"
}
}
}
}
```

索引父文档


```
PUT my_blogs/_doc/blog1
{
"title":"Learning Elasticsearch",
"content":"learning ELK is happy",
"blog_comments_relation":{
"name":"blog"
}
}

PUT my_blogs/_doc/blog2
{
"title":"Learning Hadoop",
"content":"learning Hadoop",
"blog_comments_relation":{
"name":"blog"
}
}
```

```
18
```

索引子文档

- 父文档和子文档必须存在相同的分片上
  - 确保查询 join 的性能
- 当指定文档时候，必须指定它的父文档 ID
  - 使用 route 参数来保证，分配到相同的分片

#索引子文档

```
1  PUT my_blogs/_doc/comment1?routing=blog1
2  {
3  "comment":"I am learning ELK",
4  "username":"Jack",
5  "blog_comments_relation":{
6  "name":"comment",
7  "parent":"blog1"
8  }
9  }
10
11  PUT my_blogs/_doc/comment2?routing=blog2
12  {
13  "comment":"I like Hadoop!!!!!",
14  "username":"Jack",
15  "blog_comments_relation":{
16  "name":"comment",
17  "parent":"blog2"
18  }
19  }
20
21  PUT my_blogs/_doc/comment3?routing=blog2
22  {
23  "comment":"Hello Hadoop",
```

```
"username":"Bob",
"blog_comments_relation":{
"name":"comment",
"parent":"blog2"
}
}

```

Parent / Child 所支持的查询

- 查询所有文档
- Parent Id 查询
- Has Child 查询
- Has Parent 查询

```
# 查询所有文档
POST my_blogs/_search
{}

#根据父文档ID查看
GET my_blogs/_doc/blog2

# Parent Id 查询
POST my_blogs/_search
{
"query": {
"parent_id": {
"type": "comment",
"id": "blog2"
}
}
}

# Has Child 查询,返回父文档
POST my_blogs/_search
{
"query": {
"has_child": {
"type": "comment",
"query" : {
"match": {
"username" : "Jack"
}
}
```

```
}
}
}

# Has Parent 查询，返回相关的子文档
POST my_blogs/_search
{
"query": {
"has_parent": {
"parent_type": "blog",
"query" : {
"match": {
"title" : "Learning Hadoop"
}
}
}
}
}
```

使用 has_child 查询

- 返回父文档
- 通过对子文档进行查询
  - 返回具体相关子文档的父文档
  - 父子文档在相同的分片上，因此 Join 效率高

使用 has_parent 查询

- 返回相关性的子文档

- 通过对父文档进行查询
    - 返回相关的子文档

```
POST my_blogs/_search
{
  "query": {
    "has_parent": {
      "parent_type": "blog",
      "query" : {
        "match": {
          "title" : "Learning Hadoop"
        }
      }
    }
  }
}
```

Parent Relation Name

使用 parent_id 查询

- 返回所有相关子文档
- 通过对付文档 Id 进行查询
    - 返回所有相关的子文档

```
POST my_blogs/_search
{
  "query": {
    "parent_id": {
      "type": "comment",
      "id": "blog2"
    }
  }
}
```

Parent Id

访问子文档

- 需指定父文档 routing 参数

```
GET my_blogs/_doc/comment3
```

```
{
  "_index" : "my_blogs",
  "_type" : "_doc",
  "_id" : "comment3",
  "found" : false
}
```

```
GET my_blogs/_doc/comment3?routing=blog2
```

```
{
  "_index" : "my_blogs",
  "_type" : "_doc",
  "_id" : "comment3",
  "_version" : 2,
  "_seq_no" : 5,
  "_primary_term" : 1,
  "_routing" : "blog2",
  "found" : true,
  "_source" : {
    "comment" : "Hello Hadoop??",
    "username" : "Bob",
    "blog_comments_relation" : {
      "name" : "comment",
      "parent" : "blog2"
    }
  }
}
```

```
#通过ID ， 访问子文档
GET my_blogs/_doc/comment2
#通过ID和routing ， 访问子文档
GET my_blogs/_doc/comment3?routing=blog2
```

更新子文档

- 更新子文档不会影响到父文档

```
POST my_blogs/comment3/_update?routing=blog2
{
  "doc":{
    "comment":"Hello Hadoop??"
  }
}
```

#更新子文档

```
PUT my_blogs/_doc/comment3?routing=blog2
{
"comment": "Hello Hadoop??",
"blog_comments_relation": {
"name": "comment",
"parent": "blog2"
}
}
```

嵌套对象 v.s 父子文档

Nested Object Parent / Child

优点：文档存储在一起，读取性能高、父子文档可以独立更新

缺点：更新嵌套的子文档时，需要更新整个文档、需要额外的内存去维护关系。读取性能相对差

适用场景子文档偶尔更新，以查询为主、子文档更新频繁

四、文件系统数据建模

思考一下，github中可以使用代码片段来实现数据搜索。这是如何实现的？

在github中也使用了ES来实现数据的全文搜索。其ES中有一个记录代码内容的索引，大致数据内容如下：

```
{
```

```
  "fileName" : "HelloWorld.java",
  "authName" : "baiqi",
  "authID" : 110,
  "productName" : "first-java",
  "path" : "/com/baiqi/first",
  "content" : "package com.baiqi.first; public class HelloWorld { //code... }"
}
```

我们可以在github中通过代码的片段来实现数据的搜索。也可以使用其他条件实现数据搜索。但是，如果需要使用文件路径搜索内容应该如何实现？这个时候需要为其中的字段path定义一个特殊的分词器。具体如下：

```
PUT /codes
{
"settings": {
"analysis": {
"analyzer": {
"path_analyzer" : {
"tokenizer" : "path_hierarchy"
}
}
 }
 },
 "mappings": {
 "properties": {
 "fileName" : {
 "type" : "keyword"
 },
 "authName" : {
 "type" : "text",
 "analyzer": "standard",
 "fields": {
 "keyword" : {
 "type" : "keyword"
 }
 }
 },
 "authID" : {
 "type" : "long"
 },
 "productName" : {
 "type" : "text",
 "analyzer": "standard",
 "fields": {
 "keyword" : {
```

```
"type" : "keyword"
}
}
},
"path" : {
"type" : "text",
"analyzer": "path_analyzer",
"fields": {
"keyword" : {
"type" : "keyword"
}
}
},
"content" : {
"type" : "text",
"analyzer": "standard"
}
}
}
}

PUT /codes/_doc/1
{
"fileName" : "HelloWorld.java",
"authName" : "baiqi",
"authID" : 110,
"productName" : "first-java",
"path" : "/com/baiqi/first",
"content" : "package com.baiqi.first; public class HelloWorld { // some code... }"
}

GET /codes/_search
{
"query": {
"match": {
"path": "/com"
}
}
}

GET /codes/_analyze
{
"text": "/a/b/c/d",
"field": "path"
}
```

```
79
80 ################################################################################
###########################
81 PUT /codes
82 {
83 "settings": {
84 "analysis": {
85 "analyzer": {
86 "path_analyzer" : {
87 "tokenizer" : "path_hierarchy"
88 }
89 }
90 }
91 },
92 "mappings": {
93 "properties": {
94 "fileName" : {
95 "type" : "keyword"
96 },
97 "authName" : {
98 "type" : "text",
99 "analyzer": "standard",
100 "fields": {
101 "keyword" : {
102 "type" : "keyword"
103 }
104 }
105 },
106 "authID" : {
107 "type" : "long"
108 },
109 "productName" : {
110 "type" : "text",
111 "analyzer": "standard",
112 "fields": {
113 "keyword" : {
114 "type" : "keyword"
115 }
116 }
117 },
118 "path" : {
119 "type" : "text",
120 "analyzer": "path_analyzer",
121 "fields": {
122 "keyword" : {
```

```
"type" : "text",
"analyzer": "standard"
}
}
},
"content" : {
"type" : "text",
"analyzer": "standard"
}
}
}
}

GET /codes/_search
{
"query": {
"match": {
"path.keyword": "/com"
}
}
}

GET /codes/_search
{
"query": {
"bool": {
"should": [
{
"match": {
"path": "/com"
}
},
{
"match": {
"path.keyword": "/com/baiqi"
}
}
]
}
}
}
```

参考文档：https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-pathhierarchy-tokenizer.html

五、根据关键字分页搜索

在存在大量数据时，一般我们进行查询都需要进行分页查询。例如：我们指定页码、并指定每页显示多少条数据，然后Elasticsearch返回对应页码的数据。

1、使用from和size来进行分页

在执行查询时，可以指定from（从第几条数据开始查起）和size（每页返回多少条）数据，就可以轻松完成分页。

```
| from = (page - 1) * size
```

```
POST /es_db/_doc/_search
{
"from": 0,
"size": 2,
"query": {
"match": {
"address": "广州天河"
}
}
}
```

2、使用scroll方式进行分页

前面使用from和size方式，查询在1W-5W条数据以内都是OK的，但如果数据比较多的时候，会出现性能问题。Elasticsearch做了一个限制，不允许查询的是10000条以后的数据。如果要查询1W条以后的数据，需要使用Elasticsearch中提供的scroll游标来查询。

在进行大量分页时，每次分页都需要将要查询的数据进行重新排序，这样非常浪费性能。使用scroll是将要用的数据一次性排序好，然后分批取出。性能要比from + size好得多。使用scroll查询后，排序后的数据会保持一定的时间，后续的分页查询都从该快照取数据即可。

2.1、第一次使用scroll分页查询

此处，我们让排序的数据保持1分钟，所以设置scroll为1m

```
GET /es_db/_search?scroll=1m
{
"query": {
"multi_match":{
"query":"广州长沙张三",
"fields":["address","name"]
}
},
```

```
9  "size":100
10 }
```

执行后，我们注意到，在响应结果中有一项：

"_scroll_id": "DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAZEWY2VQZXBia1JTVkdhTWkwSl9GaUYtQQ=="

后续，我们需要根据这个_scroll_id来进行查询

2.2、第二次直接使用scroll id进行查询

```
1 GET _search/scroll?scroll=1m
2 {
3 "scroll_id":"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAZoWY2VQZXBia1JTVkdhTWkwSl9GaUYtQQ=="
4 }
```

六、Elasticsearch SQL

Elasticsearch SQL允许执行类SQL的查询，可以使用REST接口、命令行或者是JDBC，都可以使用SQL来进行数据的检索和数据的聚合。

Elasticsearch SQL特点：

- 本地集成

Elasticsearch SQL是专门为Elasticsearch构建的。每个SQL查询都根据底层存储对相关节点有效执行。

- 没有额外的要求

不依赖其他的硬件、进程、运行时库，Elasticsearch SQL可以直接运行在Elasticsearch集群上

- 轻量且高效

像SQL那样简洁、高效地完成查询

1、SQL与Elasticsearch对应关系

SQL	Elasticsearch
column（列）	field（字段）
row（行）	document（文档）
table（表）	index（索引）
schema（模式）	mapping（映射）
database server（数据库服务器）	Elasticsearch集群实例

2、Elasticsearch SQL语法

```
SELECT select_expr [, ...]
[ FROM table_name ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition]
[ ORDER BY expression [ ASC | DESC ] [, ...] ]
[ LIMIT [ count ] ]
[ PIVOT ( aggregation_expr FOR column IN ( value [ [ AS ] alias ] [, ...] ) ) ]
```

目前FROM只支持单表

3、职位查询案例

3.1、查询职位索引库中的一条数据

```
format：表示指定返回的数据类型
//1.查询职位信息

GET /_sql?format=txt
{
"query":"SELECT * FROM es_db limit 1"
}
```

除了txt类型，Elasticsearch SQL还支持以下类型，

格式	描述
csv	逗号分隔符
json	JSON格式

tsv	制表符分隔符
txt	类cli表示
yaml	YAML人类可读的格式

3.2、将SQL转换为DSL

```
GET /_sql/translate
{
"query":"SELECT * FROM es_db limit 1"
}
```

结果如下：

```
{
 "size" : 1,
 "_source" : {
 "includes" : [
 "age",
 "remark",
 "sex"
 ],
 "excludes" : [ ]
 },
 "docvalue_fields" : [
 {
 "field" : "address"
 },
 {
 "field" : "book"
 },
 {
 "field" : "name"
 }
 ],
 "sort" : [
 {
 "_doc" : {
 "order" : "asc"
 }
 }
 ]
}
```

3.4、职位全文检索

3.4.1、需求

检索address包含广州和name中包含张三的用户。

3.4.2、MATCH函数

在执行全文检索时，需要使用到MATCH函数。

```
MATCH(
    field_exp,
    constant_exp
    [, options])
```

field_exp：匹配字段

constant_exp：匹配常量表达式

3.4.3、实现

```
GET /_sql?format=txt
{
"query":"select * from es_db where MATCH(address, '广州') or MATCH(name, '张三') limi
t 10"
}
```

4.4、通过Elasticsearch SQL方式实现分组统计

4.4.2、基于Elasticsearch SQL方式实现

```
GET /_sql?format=txt
{
"query":"select age, count(*) as age_cnt from es_db group by age"
}
```

这种方式要更加直观、简洁。

Elasticsearch SQL目前的一些限制

目前Elasticsearch SQL还存在一些限制。例如：不支持JOIN、不支持较复杂的子查询。所以，有一些相对复杂一些的功能，还得借助于DSL方式来实现。

七、Java API操作ES（上）

相关依赖：

```
<dependencies>
 <!-- ES的高阶的客户端API -->
 <dependency>
 <groupId>org.elasticsearch.client</groupId>
 <artifactId>elasticsearch-rest-high-level-client</artifactId>
 <version>7.6.1</version>
 </dependency>
 <dependency>
 <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.11.1</version>
  </dependency>
  <!-- 阿里巴巴出品的一款将Java对象转换为JSON、将JSON转换为Java对象的库 -->
  <dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.62</version>
  </dependency>
  <dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
  <dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>6.14.3</version>
  <scope>test</scope>
  </dependency>

</dependencies>
```

文档：05 ElasticSearch笔记.note

链接：http://note.youdao.com/noteshare?id=0c6581fe8e121e18dba9d22488cebf15&sub=467E78E05CE34249AC86C9B277D7ECA0