

一、聚合搜索技术深入

1. bucket和metric概念简介

bucket就是一个聚合搜索时的数据分组。如：销售部门有员工张三和李四，开发部门有员工王五和赵六。那么根据部门分组聚合得到结果就是两个bucket。销售部门bucket中有张三和李四，

开发部门 bucket中有王五和赵六。

metric就是对一个bucket数据执行的统计分析。如上述案例中，开发部门有2个员工，销售部门有2个员工，这就是metric。

metric有多种统计，如：求和，最大值，最小值，平均值等。

1 用一个大家容易理解的SQL语法来解释，如：select count(*) from table group by column。那么group by column分组后的每组数据就是bucket。对每个分组执行的count(*)就是metric。

2. 准备案例数据

```
1 PUT /cars
2 {
3   "mappings": {
4     "properties": {
5       "price": {
6         "type": "long"
7       },
8       "color": {
9         "type": "keyword"
10      },
11      "brand": {
12        "type": "keyword"
13      },
14      "model": {
15        "type": "keyword"
16      },
17      "sold_date": {
18        "type": "date"
19      },
20      "remark" : {
21        "type" : "text",
22        "analyzer" : "ik_max_word"
23      }
24    }
25  }
```

```
24 }  
25 }  
26 }
```

```
1 POST /cars/_bulk  
2 { "index": {} }  
3 { "price": 258000, "color": "金色", "brand": "大众", "model": "大众迈腾",  
  "sold_date": "2021-10-28", "remark": "大众中档车" }  
4 { "index": {} }  
5 { "price": 123000, "color": "金色", "brand": "大众", "model": "大众速腾",  
  "sold_date": "2021-11-05", "remark": "大众神车" }  
6 { "index": {} }  
7 { "price": 239800, "color": "白色", "brand": "标志", "model": "标志508",  
  "sold_date": "2021-05-18", "remark": "标志品牌全球上市车型" }  
8 { "index": {} }  
9 { "price": 148800, "color": "白色", "brand": "标志", "model": "标志408",  
  "sold_date": "2021-07-02", "remark": "比较大的紧凑型车" }  
10 { "index": {} }  
11 { "price": 1998000, "color": "黑色", "brand": "大众", "model": "大众辉  
   腾", "sold_date": "2021-08-19", "remark": "大众最让人肝疼的车" }  
12 { "index": {} }  
13 { "price": 218000, "color": "红色", "brand": "奥迪", "model": "奥迪A4",  
  "sold_date": "2021-11-05", "remark": "小资车型" }  
14 { "index": {} }  
15 { "price": 489000, "color": "黑色", "brand": "奥迪", "model": "奥迪A6",  
  "sold_date": "2022-01-01", "remark": "政府专用?" }  
16 { "index": {} }  
17 { "price": 1899000, "color": "黑色", "brand": "奥迪", "model": "奥迪A  
   8", "sold_date": "2022-02-12", "remark": "很贵的大A6。。。" }
```

二. 聚合操作案例

1、根据color分组统计销售数量

只执行聚合分组，不做复杂的聚合统计。在ES中最基础的聚合为terms，相当于SQL中的count。

在ES中默认为分组数据做排序，使用的是doc_count数据执行降序排列。可以使用_key元数据，根据分组后的字段数据执行不同的排序方案，也可以根据_count元数据，根据分组后的统计值执行不同的排序方案。

```
1 GET /cars/_search
```

```
2 {
3   "aggs": {
4     "group_by_color": {
5       "terms": {
6         "field": "color",
7         "order": {
8           "_count": "desc"
9         }
10      }
11    }
12  }
13 }
```

2、统计不同color车辆的平均价格

本案例先根据color执行聚合分组，在此分组的基础上，对组内数据执行聚合统计，这个组内数据的聚合统计就是metric。同样可以执行排序，因为组内有聚合统计，且对统计数据给予了命名avg_by_price，所以可以根据这个聚合统计数据字段名执行排序逻辑。

```
1 GET /cars/_search
2 {
3   "aggs": {
4     "group_by_color": {
5       "terms": {
6         "field": "color",
7         "order": {
8           "avg_by_price": "asc"
9         }
10      },
11      "aggs": {
12        "avg_by_price": {
13          "avg": {
14            "field": "price"
15          }
16        }
17      }
18    }
19  }
20 }
```

size可以设置为0，表示不返回ES中的文档，只返回ES聚合之后的数据，提高查询速度，当然如果你需要这些文档的话，也可以按照实际情况进行设置

```
1 GET /cars/_search
2 {
3   "size" : 0,
4   "aggs": {
5     "group_by_color": {
6       "terms": {
7         "field": "color"
8       },
9     "aggs": {
10      "group_by_brand" : {
11        "terms": {
12          "field": "brand",
13          "order": {
14            "avg_by_price": "desc"
15          }
16        },
17      "aggs": {
18        "avg_by_price": {
19          "avg": {
20            "field": "price"
21          }
22        }
23      }
24    }
25  }
26 }
27 }
28 }
```

3、统计不同color不同brand中车辆的平均价格

先根据color聚合分组，在组内根据brand再次聚合分组，这种操作可以称为下钻分析。

Aggs如果定义比较多，则会感觉语法格式混乱，aggs语法格式，有一个相对固定的结构，简单定义：aggs可以嵌套定义，可以水平定义。

嵌套定义称为下钻分析。水平定义就是平铺多个分组方式。

```
1 GET /index_name/type_name/_search
2 {
3   "aggs" : {
4     "定义分组名称（最外层）": {
5       "分组策略如: terms、avg、sum" : {
6         "field" : "根据哪一个字段分组",
7         "其他参数" : ""
8       },
9       "aggs" : {
10        "分组名称1" : {},
11        "分组名称2" : {}
12      }
13    }
14  }
15 }
```

```
1 GET /cars/_search
2 {
3   "aggs": {
4     "group_by_color": {
5       "terms": {
6         "field": "color",
7         "order": {
8           "avg_by_price_color": "asc"
9         }
10      },
11      "aggs": {
12        "avg_by_price_color" : {
13          "avg": {
14            "field": "price"
15          }
16        },
17        "group_by_brand" : {
18          "terms": {
19            "field": "brand",
20            "order": {
21              "avg_by_price_brand": "desc"
22            }
23          }
24        }
25      }
26    }
27  }
```

```
23 },
24 "aggs": {
25   "avg_by_price_brand": {
26     "avg": {
27       "field": "price"
28     }
29   }
30 }
31 }
32 }
33 }
34 }
35 }
```

4、统计不同color中的最大和最小价格、总价

```
1 GET /cars/_search
2 {
3   "aggs": {
4     "group_by_color": {
5       "terms": {
6         "field": "color"
7       },
8       "aggs": {
9         "max_price": {
10          "max": {
11            "field": "price"
12          }
13        },
14        "min_price" : {
15          "min": {
16            "field": "price"
17          }
18        },
19        "sum_price" : {
20          "sum": {
21            "field": "price"
22          }
23        }
24      }
25    }
26  }
```

```
24 }  
25 }  
26 }  
27 }
```

在常见的业务常见中，聚合分析，最常用的种类就是统计数量，最大，最小，平均，总计等。通常占有聚合业务中的60%以上的比例，小型项目中，甚至占比85%以上。

三、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。

```
1 PUT /user_index  
2 {  
3   "mappings": {  
4     "properties": {  
5       "login_name" : {  
6         "type" : "keyword"  
7       },  
8       "age " : {  
9         "type" : "short"  
10      },  
11      "address" : {  
12        "properties": {  
13          "province" : {  
14            "type" : "keyword"  
15          },  
16          "city" : {  
17            "type" : "keyword"  
18          },  
19          "street" : {  
20            "type" : "keyword"  
21          }  
22        }  
23      }  
24    }
```

```
25 }  
26 }
```

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。

```
1  PUT /user_index/_doc/1  
2  {  
3    "login_name" : "jack",  
4    "age" : 25,  
5    "address" : [  
6      {  
7        "province" : "北京",  
8        "city" : "北京",  
9        "street" : "枫林三路"  
10     },  
11     {  
12       "province" : "天津",  
13       "city" : "天津",  
14       "street" : "华夏路"  
15     }  
16   ]  
17 }  
18 PUT /user_index/_doc/2  
19 {  
20   "login_name" : "rose",  
21   "age" : 21,  
22   "address" : [  
23     {  
24       "province" : "河北",  
25       "city" : "廊坊",  
26       "street" : "燕郊经济开发区"  
27     },  
28     {  
29       "province" : "天津",  
30       "city" : "天津",  
31       "street" : "华夏路"  
32     }  
33   ]
```


执行的搜索应该如下：

```
1 GET /user_index/_search
2 {
3   "query": {
4     "bool": {
5       "must": [
6         {
7           "match": {
8             "address.province": "北京"
9         }
10      },
11     {
12       "match": {
13         "address.city": "天津"
14       }
15     }
16   ]
17 }
18 }
19 }
```

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：

```
1 PUT /user_index
2 {
3   "mappings": {
4     "properties": {
5       "login_name" : {
6         "type" : "keyword"
7       },
8       "age" : {
```

```
9  "type" : "short"
10 },
11 "address" : {
12   "type": "nested",
13   "properties": {
14     "province" : {
15       "type" : "keyword"
16     },
17     "city" : {
18       "type" : "keyword"
19     },
20     "street" : {
21       "type" : "keyword"
22     }
23   }
24 }
25 }
26 }
27 }
```

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：

```
1  GET /user_index/_search
2  {
3    "query": {
4      "bool": {
5        "must": [
6          {
7            "nested": {
8              "path": "address",
9              "query": {
10               "bool": {
11                 "must": [
12                   {
13                     "match": {
14                       "address.province": "北京"
15                     }
16                   },
17                   {
18                     "match": {
```

```
19  "address.city": "天津"
20  }
21  }
22  ]
23  }
24  }
25  }
26  }
27  ]
28  }
29  }
30  }
```

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）

```
1  {
2    "login_name" : "jack",
3    "address.province" : [ "北京", "天津" ],
4    "address.city" : [ "北京", "天津" ]
5    "address.street" : [ "枫林三路", "华夏路" ]
6  }
```

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。

```
1  {
2    "login_name" : "jack"
3  }
4  {
5    "address.province" : "北京",
6    "address.city" : "北京",
7    "address.street" : "枫林三路"
8  }
```

```
9 {  
10   "address.province" : "天津",  
11   "address.city" : "天津",  
12   "address.street" : "华夏路",  
13 }
```

3、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。

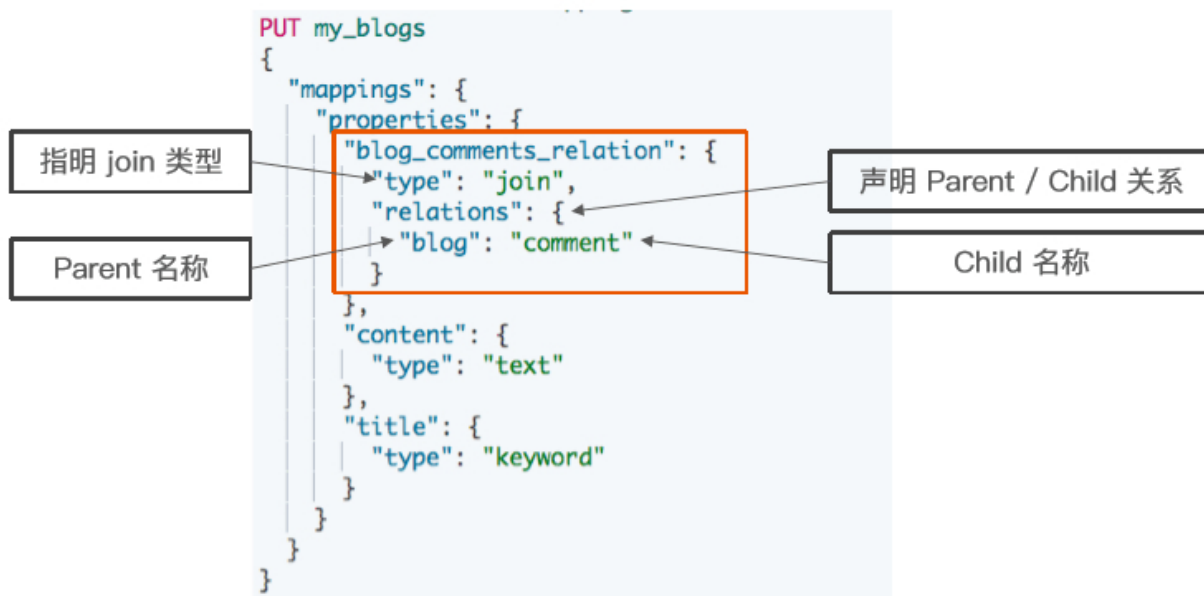
要点：父子关系元数据映射，用于确保查询时候的高性能，但是有一个限制，就是父子数据必须存在于一个shard中

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



```
1 DELETE my_blogs
2 # 设定 Parent/Child Mapping
3 PUT my_blogs
4 {
5
6   "mappings": {
7     "properties": {
8       "blog_comments_relation": {
9         "type": "join",
10        "relations": {
11          "blog": "comment"
12        }
13      },
14      "content": {
15        "type": "text"
16      },
17      "title": {
18        "type": "keyword"
19      }
20    }
21  }
22 }
```

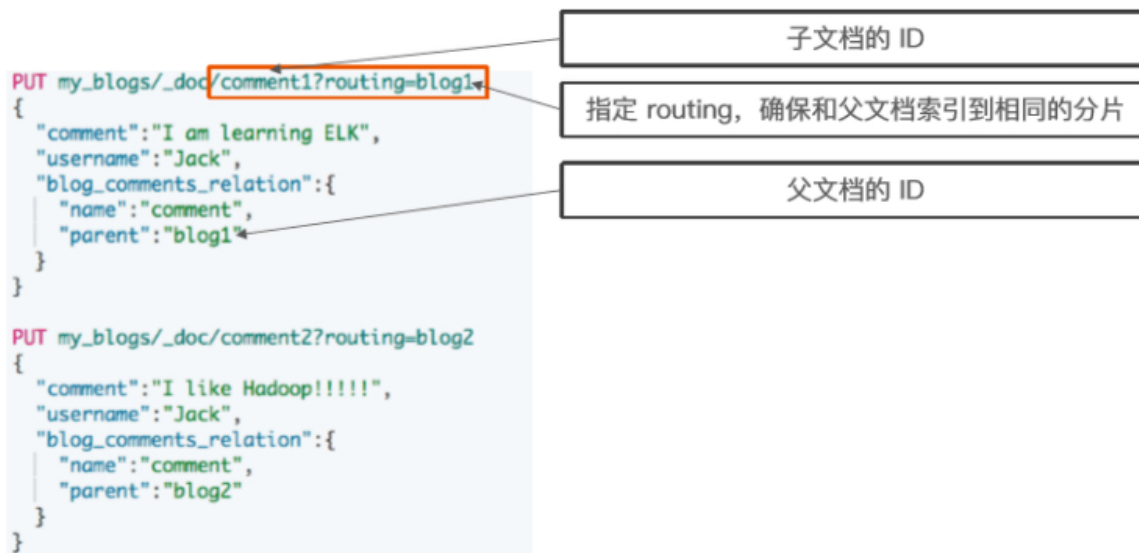
索引父文档



```
1 PUT my_blogs/_doc/blog1
2 {
3   "title": "Learning Elasticsearch",
4   "content": "learning ELK is happy",
5   "blog_comments_relation": {
6     "name": "blog"
7   }
8 }
9
10 PUT my_blogs/_doc/blog2
11 {
12   "title": "Learning Hadoop",
13   "content": "learning Hadoop",
14   "blog_comments_relation": {
15     "name": "blog"
16   }
17 }
```

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",
24   "username": "Bob",
25   "blog_comments_relation": {
26     "name": "comment",
```

```
27 "parent":"blog2"
28 }
29 }
```

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

```
1 # 查询所有文档
2 POST my_blogs/_search
3 {}
4
5 #根据父文档ID查看
6 GET my_blogs/_doc/blog2
7
8 # Parent Id 查询
9 POST my_blogs/_search
10 {
11   "query": {
12     "parent_id": {
13       "type": "comment",
14       "id": "blog2"
15     }
16   }
17 }
18
19 # Has Child 查询,返回父文档
20 POST my_blogs/_search
21 {
22   "query": {
23     "has_child": {
24       "type": "comment",
25       "query" : {
26         "match": {
27           "username" : "Jack"
28         }
29       }
30     }
31   }
32 }
```



```
29  }
30  }
31  }
32  }
33
34  # Has Parent 查询，返回相关的子文档
35  POST my_blogs/_search
36  {
37    "query": {
38      "has_parent": {
39        "parent_type": "blog",
40        "query" : {
41          "match": {
42            "title" : "Learning Hadoop"
43          }
44        }
45      }
46    }
47  }
```

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

```
POST my_blogs/_search
{
  "query": {
    "has_child": {
      "type": "comment",
      "query": {
        "match": {
          "username": "Jack"
        }
      }
    }
  }
}
```

Child Relation Name

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
- 通过对父文档 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"
    }
  }
}
```

- 1 #通过ID ， 访问子文档
- 2 GET my_blogs/_doc/comment2
- 3
- 4 #通过ID和routing ， 访问子文档
- 5 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

```
1 PUT my_blogs/_doc/comment3?routing=blog2
2 {
3   "comment": "Hello Hadoop??",
4   "blog_comments_relation": {
5     "name": "comment",
6     "parent": "blog2"
7   }
8 }
```

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

4、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：

```
1 {
2   "fileName" : "HelloWorld.java",
3   "authName" : "baiqi",
4   "authID" : 110,
5   "productName" : "first-java",
6   "path" : "/com/baiqi/first",
```

```
7   "content" : "package com.baiqi.first; public class HelloWorld { //cod
e... }"
8 }
```

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：

```
1 PUT /codes
2 {
3   "settings": {
4     "analysis": {
5       "analyzer": {
6         "path_analyzer" : {
7           "tokenizer" : "path_hierarchy"
8         }
9       }
10    }
11  },
12  "mappings": {
13    "properties": {
14      "fileName" : {
15        "type" : "keyword"
16      },
17      "authName" : {
18        "type" : "text",
19        "analyzer": "standard",
20        "fields": {
21          "keyword" : {
22            "type" : "keyword"
23          }
24        }
25      },
26      "authID" : {
27        "type" : "long"
28      },
29      "productName" : {
30        "type" : "text",
31        "analyzer": "standard",
32        "fields": {
```

```
33 "keyword" : {
34 "type" : "keyword"
35 }
36 }
37 },
38 "path" : {
39 "type" : "text",
40 "analyzer": "path_analyzer",
41 "fields": {
42 "keyword" : {
43 "type" : "keyword"
44 }
45 }
46 },
47 "content" : {
48 "type" : "text",
49 "analyzer": "standard"
50 }
51 }
52 }
53 }
54
55 PUT /codes/_doc/1
56 {
57 "fileName" : "HelloWorld.java",
58 "authName" : "baiqi",
59 "authID" : 110,
60 "productName" : "first-java",
61 "path" : "/com/baiqi/first",
62 "content" : "package com.baiqi.first; public class HelloWorld { // some
code... }"
63 }
64
65 GET /codes/_search
66 {
67 "query": {
68 "match": {
69 "path": "/com"
70 }
71 }
```

```
72 }
73
74 GET /codes/_analyze
75 {
76   "text": "/a/b/c/d",
77   "field": "path"
78 }
79
80 #####
81 PUT /codes
82 {
83   "settings": {
84     "analysis": {
85       "analyzer": {
86         "path_analyzer" : {
87           "tokenizer" : "path_hierarchy"
88         }
89       }
90     },
91     "mappings": {
92       "properties": {
93         "fileName" : {
94           "type" : "keyword"
95         },
96         "authName" : {
97           "type" : "text",
98           "analyzer": "standard",
99           "fields": {
100             "keyword" : {
101               "type" : "keyword"
102             }
103           }
104         },
105         "authID" : {
106           "type" : "long"
107         },
108         "productName" : {
109           "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" : {
123       "type" : "text",
124       "analyzer": "standard"
125     }
126   }
127 },
128 "content" : {
129   "type" : "text",
130   "analyzer": "standard"
131 }
132 }
133 }
134 }
135
136 GET /codes/_search
137 {
138   "query": {
139     "match": {
140       "path.keyword": "/com"
141     }
142   }
143 }
144
145 GET /codes/_search
146 {
147   "query": {
148     "bool": {
149       "should": [
150   {
```



```
151 "match": {
152   "path": "/com"
153 }
154 },
155 {
156   "match": {
157     "path.keyword": "/com/baiqi"
158   }
159 }
160 ]
161 }
162 }
163 }
```

参考文档：

<https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-pathhierarchy-tokenizer.html>

四、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

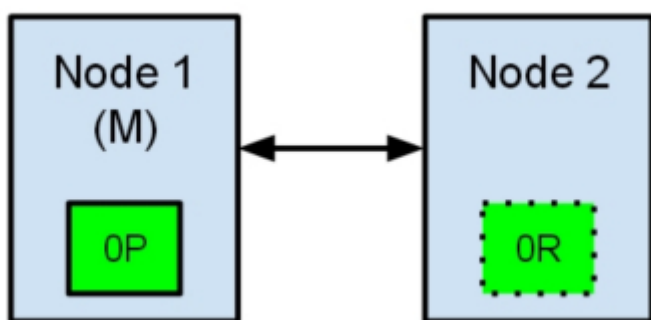
集群脑裂是什么？

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。

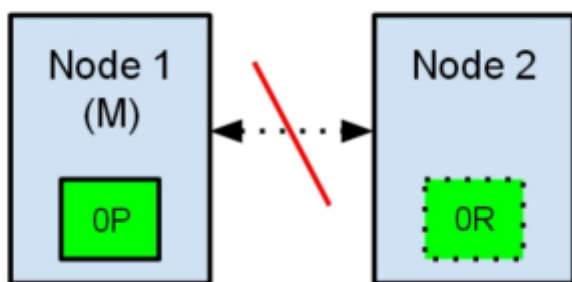
但是因为master是集群中非常重要的一个角色，主宰了集群状态的维护，以及shard的分配，因此如果有两个master，可能会导致数据异常。

如：



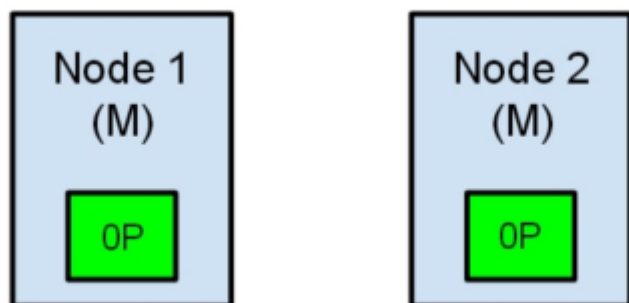
节点1在启动时被选举为主节点并保存主分片标记为0P，而节点2保存复制分片标记为0R

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Node2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：
 $\text{master候选节点数量} / 2 + 1$ 。

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。

五、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

```
1 query "hello world" --> hello / world / hello & world
2
3 bool --> must/must not/should --> 过滤 --> 包含 / 不包含 / 可能包含
4
```

5 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

```
1 搜索请求: hello world
2
3
4
5 doc1: hello you, and world is very good
6
7 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

```
1 搜索请求: hello world
2
3
4
5 doc1: hello, tuling is very good
6
7 doc2: hi world, how are you
```

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" }
```

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