

主讲老师：Fox

课前须知：

本节课会先补第四节课的内容，第四节课没听的同学补一下。

ES要掌握什么：

1. 使用：查询语法，聚合操作语法，理解相关性算分（文档匹配度）
2. 优化：文档建模，语法选择，ES集群架构优化，数据处理

有道云笔记链接

- 1 文档：5. Elasticsearch高级功能详解与原理剖...
- 2 链接：<http://note.youdao.com/noteshare?id=c6fd46d242de4a7c3ba43ba9156c2293&sub=00A4B45C1A8947E8A126D788F4D6D8C3>

Elasticsearch中如何处理关联关系

对象类型

嵌套对象(Nested Object)

父子关联关系(Parent / Child)

嵌套文档 VS 父子文档

Ingest Pipeline & Painless Script

Pipeline & Processor

ElasticSearch数据建模最佳实践

建模建议1：如何处理关联关系

建模建议2: 避免过多字段

建模建议3：避免正则，通配符，前缀查询

建模建议4：避免空值引起的聚合不准

建模建议5: 为索引的Mapping加入Meta 信息

Elasticsearch中如何处理关联关系

关系型数据库范式化（Normalize）设计的主要目标是减少不必要的更新，往往会带来一些副作用：

- 一个完全范式化设计的数据库会经常面临“查询缓慢”的问题。数据库越范式化，就需要Join越多的表；
- 范式化节省了存储空间，但是存储空间已经变得越来越便宜；
- 范式化简化了更新，但是数据读取操作可能更多。

反范式化(Denormalize)的设计不使用关联关系，而是在文档中保存冗余的数据拷贝。

- **优点:** 无需处理Join操作，数据读取性能好。Elasticsearch可以通过压缩_source 字段，减少磁盘空间的开销
- **缺点:** 不适合在数据频繁修改的场景。一条数据的改动，可能会引起很多数据的更新

关系型数据库，一般会考虑Normalize 数据；在Elasticsearch，往往考虑Denormalize 数据。

Elasticsearch并不擅长处理关联关系，一般会采用以下四种方法处理关联：

- 对象类型
- 嵌套对象(Nested Object)
- 父子关联关系(Parent / Child)
- 应用端关联

对象类型

案例1：博客作者信息变更

对象类型：

- 在每一博客的文档中都保留作者的信息
- 如果作者信息发生变化，需要修改相关的博客文档

```
1 DELETE blog
2 # 设置blog的 Mapping
3 PUT /blog
4 {
5   "mappings": {
6     "properties": {
7       "content": {
8         "type": "text"
```

```
9   },
10  "time": {
11    "type": "date"
12  },
13  "user": {
14    "properties": {
15      "city": {
16        "type": "text"
17      },
18      "userid": {
19        "type": "long"
20      },
21      "username": {
22        "type": "keyword"
23      }
24    }
25  }
26 }
27 }
28 }
29
30 # 插入一条 blog信息
31 PUT /blog/_doc/1
32 {
33   "content":"I like Elasticsearch",
34   "time":"2022-01-01T00:00:00",
35   "user":{
36     "userid":1,
37     "username":"Fox",
38     "city":"Changsha"
39   }
40 }
41
42
43 # 查询 blog信息
44 POST /blog/_search
45 {
46   "query": {
47     "bool": {
48       "must": [
```

```
49  {"match": {"content": "Elasticsearch"}},
50  {"match": {"user.username": "Fox"}}
51  ]
52  }
53  }
54  }
```

案例2：包含对象数组的文档

```
1  DELETE /my_movies
2
3  # 电影的Mapping信息
4  PUT /my_movies
5  {
6    "mappings" : {
7      "properties" : {
8        "actors" : {
9          "properties" : {
10           "first_name" : {
11             "type" : "keyword"
12           },
13           "last_name" : {
14             "type" : "keyword"
15           }
16         }
17       },
18       "title" : {
19         "type" : "text",
20         "fields" : {
21           "keyword" : {
22             "type" : "keyword",
23             "ignore_above" : 256
24           }
25         }
26       }
27     }
28   }
29 }
30
31
32 # 写入一条电影信息
```

```

33 POST /my_movies/_doc/1
34 {
35   "title":"Speed",
36   "actors":[
37     {
38       "first_name":"Keanu",
39       "last_name":"Reeves"
40     },
41
42     {
43       "first_name":"Dennis",
44       "last_name":"Hopper"
45     }
46
47   ]
48 }
49
50 # 查询电影信息
51 POST /my_movies/_search
52 {
53   "query": {
54     "bool": {
55       "must": [
56         {"match": {"actors.first_name": "Keanu"}},
57         {"match": {"actors.last_name": "Hopper"}}
58       ]
59     }
60   }
61
62 }

```

```


# 查询电影信息
POST my_movies/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"actors.first_name": "Keanu"}},
        {"match": {"actors.last_name": "Hopper"}}
      ]
    }
  }
}

```

```

17 {
18   "_index" : "my_movies",
19   "_type" : "_doc",
20   "_id" : "1",
21   "_score" : 0.723315,
22   "_source" : {
23     "title" : "Speed",
24     "actors" : [
25       {
26         "first_name" : "Keanu",
27         "last_name" : "Reeves"
28       },
29       {
30         "first_name" : "Dennis",
31         "last_name" : "Hopper"
32       }
33     ]
34   }

```



思考：为什么会搜到不需要的结果？

存储时，内部对象的边界并没有考虑在内,JSON格式被处理成扁平式键值对的结构。当对多个字段进行查询时，导致了意外的搜索结果。可以用Nested Data Type解决这个问题。

```
1 "title":"Speed"
2 "actor".first_name: ["Keanu","Dennis"]
3 "actor".last_name: ["Reeves","Hopper"]
```

嵌套对象(Nested Object)

什么是Nested Data Type

- **Nested数据类型**: 允许对象数组中的对象被独立索引
- 使用nested 和properties 关键字，将所有actors索引到多个分隔的文档
- 在内部, Nested文档会被保存在两个Lucene文档中，在查询时做Join处理

```
1 DELETE /my_movies
2 # 创建 Nested 对象 Mapping
3 PUT /my_movies
4 {
5   "mappings" : {
6     "properties" : {
7       "actors" : {
8         "type": "nested",
9         "properties" : {
10          "first_name" : {"type" : "keyword"},
11          "last_name" : {"type" : "keyword"}
12        },
13        "title" : {
14          "type" : "text",
15          "fields" : {"keyword":{"type":"keyword","ignore_above":256}}
16        }
17      }
18    }
19  }
20
21 POST /my_movies/_doc/1
22 {
23   "title":"Speed",
24   "actors":[
25     {
26       "first_name":"Keanu",
27       "last_name":"Reeves"
```

```
28  },
29
30  {
31    "first_name": "Dennis",
32    "last_name": "Hopper"
33  }
34
35  ]
36 }
37
38 # Nested 查询
39 POST /my_movies/_search
40 {
41   "query": {
42     "bool": {
43       "must": [
44         {"match": {"title": "Speed"}},
45         {
46           "nested": {
47             "path": "actors",
48             "query": {
49               "bool": {
50                 "must": [
51                   {"match": {
52                     "actors.first_name": "Keanu"
53                   }},
54
55                   {"match": {
56                     "actors.last_name": "Hopper"
57                   }}
58                 ]
59               }
60             }
61           }
62         }
63       ]
64     }
65   }
66 }
67
```

```

68 # Nested Aggregation
69 POST /my_movies/_search
70 {
71   "size": 0,
72   "aggs": {
73     "actors": {
74       "nested": {
75         "path": "actors"
76       },
77       "aggs": {
78         "actor_name": {
79           "terms": {
80             "field": "actors.first_name",
81             "size": 10
82           }
83         }
84       }
85     }
86   }
87 }
88
89
90 # 普通 aggregation不工作
91 POST /my_movies/_search
92 {
93   "size": 0,
94   "aggs": {
95     "NAME": {
96       "terms": {
97         "field": "actors.first_name",
98         "size": 10
99       }
100     }
101   }
102 }

```

父子关联关系(Parent / Child)

- 对象和Nested对象的局限性: 每次更新, 可能需要重新索引整个对象(包括根对象和嵌套对象)
- 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, 可以通过维护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子文档。子文档被添加, 更新或者删除也不会影响到父文档和其他的子文档

设定 Parent/Child Mapping

```
1 DELETE /my_blogs
2
3 # 设定 Parent/Child Mapping
4 PUT /my_blogs
5 {
6   "settings": {
7     "number_of_shards": 2
8   },
9   "mappings": {
10    "properties": {
11      "blog_comments_relation": {
12        "type": "join",
13        "relations": {
14          "blog": "comment"
15        }
16      },
17      "content": {
18        "type": "text"
19      },
20      "title": {
21        "type": "keyword"
22      }
23    }
24  }
25 }
```

```

"properties": {
  "blog_comments_relation": {
    "type": "join",
    "relations": {
      "blog": "comment"
    }
  }
},

```

指明join类

child名称

Parent名称

索引父文档

```

1 #索引父文档
2 PUT /my_blogs/_doc/blog1
3 {
4   "title":"Learning Elasticsearch",
5   "content":"learning ELK ",
6   "blog_comments_relation":{
7     "name":"blog"
8   }
9 }
10
11 #索引父文档
12 PUT /my_blogs/_doc/blog2
13 {
14   "title":"Learning Hadoop",
15   "content":"learning Hadoop",
16   "blog_comments_relation":{
17     "name":"blog"
18   }
19 }

```

```

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

```

父文档id

声明文档的类型

索引子文档

```

1 #索引子文档
2 PUT /my_blogs/_doc/comment1?routing=blog1
3 {
4   "comment":"I am learning ELK",
5   "username":"Jack",
6   "blog_comments_relation":{
7     "name":"comment",
8     "parent":"blog1"
9   }
10 }
11
12 #索引子文档
13 PUT /my_blogs/_doc/comment2?routing=blog2
14 {
15   "comment":"I like Hadoop!!!!",
16   "username":"Jack",
17   "blog_comments_relation":{
18     "name":"comment",
19     "parent":"blog2"
20   }
21 }
22
23 #索引子文档
24 PUT /my_blogs/_doc/comment3?routing=blog2
25 {
26   "comment":"Hello Hadoop",
27   "username":"Bob",
28   "blog_comments_relation":{
29     "name":"comment",
30     "parent":"blog2"
31   }
32 }

```

PUT my_blogs/_doc/comment1?routing=blog1

{

"comment":"I am learning ELK",

"username":"Jack",

"blog_comments_relation":{

"name":"comment",

"parent":"blog1"

}

}

子文档id

父文档id

指定routing,确保和父文档索引到相同的分片

注意:

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

查询

```
1 # 查询所有文档
2 POST /my_blogs/_search
3
4 #根据父文档ID查看
5 GET /my_blogs/_doc/blog2
6
7 # Parent Id 查询
8 POST /my_blogs/_search
9 {
10   "query": {
11     "parent_id": {
12       "type": "comment",
13       "id": "blog2"
14     }
15   }
16 }
17
18 # Has Child 查询,返回父文档
19 POST /my_blogs/_search
20 {
21   "query": {
22     "has_child": {
23       "type": "comment",
24       "query" : {
25         "match": {
26           "username" : "Jack"
27         }
28       }
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 }
48
49 #通过ID，访问子文档
50 GET /my_blogs/_doc/comment3
51 #通过ID和routing，访问子文档
52 GET /my_blogs/_doc/comment3?routing=blog2
53
54 #更新子文档
55 PUT /my_blogs/_doc/comment3?routing=blog2
56 {
57   "comment": "Hello Hadoop??",
58   "blog_comments_relation": {
59     "name": "comment",
60     "parent": "blog2"
61   }
62 }

```

嵌套文档 VS 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

Ingest Pipeline & Painless Script

应用场景：修复与增强写入数据

案例

需求：Tags字段中，逗号分隔的文本应该是数组，而不是一个字符串。后期需要对Tags进行Aggregation统计

```
1 #Blog数据，包含3个字段，tags用逗号间隔
2 PUT tech_blogs/_doc/1
3 {
4   "title":"Introducing big data.....",
5   "tags":"hadoop,elasticsearch,spark",
6   "content":"You konw, for big data"
7 }
```

Ingest Node

Elasticsearch 5.0后，引入的一种新的节点类型。默认配置下，每个节点都是Ingest Node：

- 具有预处理数据的能力，可拦截Index或 Bulk API的请求
- 对数据进行转换，并重新返回给Index或 Bulk API

无需Logstash，就可以进行数据的预处理，例如：

- 为某个字段设置默认值;重命名某个字段的字段名;对字段值进行Split 操作
- 支持设置Painless脚本，对数据进行更加复杂的加工

Pipeline & Processor

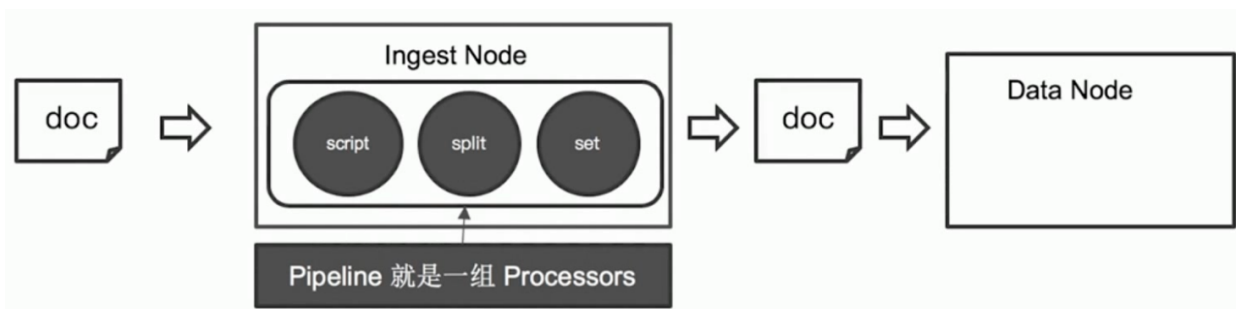
- Pipeline ——管道会对通过的数据(文档)，按照顺序进行加工
- Processor——Elasticsearch 对一些加工的行为进行了抽象包装
- Elasticsearch 有很多内置的Processors，也支持通过插件的方式，实现自己的Processor

一些内置的Processors

<https://www.elastic.co/guide/en/elasticsearch/reference/7.17/ingest-processors.html>

- Split Processor：将给定字段值分成一个数组
- Remove / Rename Processor：移除一个重命名字段
- Append：为商品增加一个新的标签

- Convert: 将商品价格, 从字符串转换成float 类型
- Date / JSON: 日期格式转换, 字符串转JSON对象
- Date Index Name Processor : 将通过该处理器的文档, 分配到指定时间格式的索引中
- Fail Processor : 一旦出现异常, 该Pipeline 指定的错误信息能返回给用户
- Foreach Process : 数组字段, 数组的每个元素都会使用到一个相同的处理器
- Grok Processor : 日志的日期格式切割)
- Gsub / Join / Split : 字符串替换 | 数组转字符串/字符串转数组
- Lowercase / upcase : 大小写转换



```

1 # 测试split tags
2 POST _ingest/pipeline/_simulate
3 {
4   "pipeline": {
5     "description": "to split blog tags",
6     "processors": [
7       {
8         "split": {
9           "field": "tags",
10          "separator": ",",
11        }
12      }
13    ],
14  },
15  "docs": [
16    {
17      "_index": "index",
18      "_id": "id",
19      "_source": {
20        "title": "Introducing big data.....",
21        "tags": "hadoop,elasticsearch,spark",
22        "content": "You konw, for big data"
23      }
24    }
25  ]
26 }

```

```
23  }
24  },
25  {
26    "_index": "index",
27    "_id": "idxx",
28    "_source": {
29      "title": "Introducing cloud computing",
30      "tags": "openstack,k8s",
31      "content": "You konw, for cloud"
32    }
33  }
34  ]
35  }
36
37  #同时为文档，增加一个字段。blog查看量
38  POST _ingest/pipeline/_simulate
39  {
40    "pipeline": {
41      "description": "to split blog tags",
42      "processors": [
43        {
44          "split": {
45            "field": "tags",
46            "separator": ",",
47          }
48        },
49
50        {
51          "set": {
52            "field": "views",
53            "value": 0
54          }
55        }
56      ]
57    },
58
59    "docs": [
60      {
61        "_index": "index",
62        "_id": "id",
```



```

63  "_source":{
64    "title":"Introducing big data.....",
65    "tags":"hadoop,elasticsearch,spark",
66    "content":"You konw, for big data"
67  }
68 },
69 {
70   "_index":"index",
71   "_id":"idxx",
72   "_source":{
73     "title":"Introducing cloud computering",
74     "tags":"openstack,k8s",
75     "content":"You konw, for cloud"
76   }
77 }
78
79 ]
80 }

```

创建pipeline

```

1  # 为ES添加一个 Pipeline
2  PUT _ingest/pipeline/blog_pipeline
3  {
4    "description": "a blog pipeline",
5    "processors": [
6      {
7        "split": {
8          "field": "tags",
9          "separator": ",",
10       }
11     },
12
13     {
14       "set":{
15         "field": "views",
16         "value": 0
17       }
18     }
19   ]
20 }

```

```
21
22 #查看Pipeline
23 GET _ingest/pipeline/blog_pipeline
24
```

使用pipeline更新数据

```
1
2 #不使用pipeline更新数据
3 PUT tech_blogs/_doc/1
4 {
5   "title": "Introducing big data.....",
6   "tags": "hadoop,elasticsearch,spark",
7   "content": "You konw, for big data"
8 }
9
10 #使用pipeline更新数据
11 PUT tech_blogs/_doc/2?pipeline=blog_pipeline
12 {
13   "title": "Introducing cloud computing",
14   "tags": "openstack,k8s",
15   "content": "You konw, for cloud"
16 }
```

借助update_by_query更新已存在的文档

```
1 #update_by_query 会导致错误
2 POST tech_blogs/_update_by_query?pipeline=blog_pipeline
3 {
4 }
5
6 #增加update_by_query的条件
7 POST tech_blogs/_update_by_query?pipeline=blog_pipeline
8 {
9   "query": {
10     "bool": {
11       "must_not": {
12         "exists": {
13           "field": "views"
14         }
15       }
16     }
17 }
```

```

18 }
19
20 GET tech_blogs/_search

```

Ingest Node VS Logstash

	Logstash	Ingest Node
数据输入与输出	支持从不同的数据源读取，并写入不同的数据源	支持从ES REST API获取数据，并且写入Elasticsearch
数据缓冲	实现了简单的数据队列，支持重写	不支持缓冲
数据处理	支持大量的插件，也支持定制开发	内置的插件，可以开发Plugin进行扩展(Plugin更新需要重启)
配置和使用	增加了一定的架构复杂度	无需额外部署

Painless

自Elasticsearch 5.x后引入，专门为Elasticsearch 设计，扩展了Java的语法。6.0开始，ES 只支持 Painless。Groovy, JavaScript和 Python 都不再支持。Painless支持所有Java 的数据类型及Java API子集。

Painless Script具备以下特性：

- 高性能/安全
- 支持显示类型或者动态定义类型

Painless的用途：

- 可以对文档字段进行加工处理
 - 更新或删除字段，处理数据聚合操作
 - Script Field:对返回的字段提前进行计算
 - Function Score:对文档的算分进行处理
- 在Ingest Pipeline中执行脚本
- 在Reindex API, Update By Query时，对数据进行处理

通过Painless脚本访问字段

上下文	语法
Ingestion	ctx.field_name
Update	ctx._source.field_name

Search & Aggregation	doc["field_name"]
----------------------	-------------------

测试

```
1 # 增加一个 Script Processor
2 POST _ingest/pipeline/_simulate
3 {
4   "pipeline": {
5     "description": "to split blog tags",
6     "processors": [
7       {
8         "split": {
9           "field": "tags",
10          "separator": ",",
11        }
12      },
13      {
14        "script": {
15          "source": ""
16          if(ctx.containsKey("content")){
17            ctx.content_length = ctx.content.length();
18          }else{
19            ctx.content_length=0;
20          }
21
22
23          ""
24        }
25      },
26
27      {
28        "set":{
29          "field": "views",
30          "value": 0
31        }
32      }
33    ]
34  },
35
36  "docs": [
37    {
```

```
38  "_index":"index",
39  "_id":"id",
40  "_source":{
41    "title":"Introducing big data.....",
42    "tags":"hadoop,elasticsearch,spark",
43    "content":"You konw, for big data"
44  }
45 },
46
47
48 {
49   "_index":"index",
50   "_id":"idx",
51   "_source":{
52     "title":"Introducing cloud computing",
53     "tags":"openstack,k8s",
54     "content":"You konw, for cloud"
55   }
56 }
57
58 ]
59 }
60
61 DELETE tech_blogs
62 PUT tech_blogs/_doc/1
63 {
64   "title":"Introducing big data.....",
65   "tags":"hadoop,elasticsearch,spark",
66   "content":"You konw, for big data",
67   "views":0
68 }
69
70 POST tech_blogs/_update/1
71 {
72   "script": {
73     "source": "ctx._source.views += params.new_views",
74     "params": {
75       "new_views":100
76     }
77 }
```

```
78 }
79
80 # 查看views计数
81 POST tech_blogs/_search
82
83
84
85 #保存脚本在 Cluster State
86 POST _scripts/update_views
87 {
88   "script":{
89     "lang": "painless",
90     "source": "ctx._source.views += params.new_views"
91   }
92 }
93
94 POST tech_blogs/_update/1
95 {
96   "script": {
97     "id": "update_views",
98     "params": {
99       "new_views":1000
100   }
101 }
102 }
103
104
105 GET tech_blogs/_search
106 {
107   "script_fields": {
108     "rnd_views": {
109       "script": {
110         "lang": "painless",
111         "source": ""
112         java.util.Random rnd = new Random();
113         doc['views'].value+rnd.nextInt(1000);
114         ""
115       }
116     }
117   },
118   "query": {
```

```
119  "match_all": {}
120  }
121  }
```

脚本缓存

脚本编译的开销较大，Elasticsearch会将脚本编译后缓存在Cache 中

- Inline scripts和 Stored Scripts都会被缓存
- 默认缓存100个脚本

参数	说明
script.cache.max_size	设置最大缓存数
script.cache.expire	设置缓存超时
script.max_compilations_rate	默认5分钟最多75次编译 (75/5m)

ElasticSearch数据建模最佳实践

建模建议1：如何处理关联关系

- Object: 优先考虑反范式（Denormalization）
- Nested: 当数据包含多数值对象，同时有查询需求
- Child/Parent: 关联文档更新非常频繁时

建模建议2: 避免过多字段

- 一个文档中，最好避免大量的字段
 - 过多的字段数不容易维护
 - Mapping 信息保存在Cluster State 中，数据量过大，对集群性能会有影响
 - 删除或者修改数据需要reindex
- 默认最大字段数是1000，可以设置index.mapping.total_fields.limit限定最大字段数。

思考：什么原因会导致文档中有成百上千的字段？

生产环境中，尽量不要打开 Dynamic，可以使用Strict控制新增字段的加入

- true：未知字段会被自动加入
- false：新字段不会被索引，但是会保存在_source

- **strict**：新增字段不会被索引，文档写入失败

对于多属性的字段，比如cookie，商品属性，可以考虑使用Nested

建模建议3：避免正则，通配符，前缀查询

正则，通配符查询，前缀查询属于Term查询，但是性能不够好。特别是将通配符放在开头，会导致性能的灾难

案例：针对版本号的搜索

```
1 # 将字符串转对象
2 PUT softwares/
3 {
4   "mappings": {
5     "properties": {
6       "version": {
7         "properties": {
8           "display_name": {
9             "type": "keyword"
10          },
11         "hot_fix": {
12           "type": "byte"
13         },
14         "marjor": {
15           "type": "byte"
16         },
17         "minor": {
18           "type": "byte"
19         }
20       }
21     }
22   }
23 }
24 }
25
26
27 #通过 Inner Object 写入多个文档
28 PUT softwares/_doc/1
29 {
30   "version":{
31     "display_name":"7.1.0",
32     "marjor":7,
```



```
33   "minor":1,
34   "hot_fix":0
35 }
36
37 }
38
39 PUT softwares/_doc/2
40 {
41   "version":{
42     "display_name":"7.2.0",
43     "marjor":7,
44     "minor":2,
45     "hot_fix":0
46   }
47 }
48
49 PUT softwares/_doc/3
50 {
51   "version":{
52     "display_name":"7.2.1",
53     "marjor":7,
54     "minor":2,
55     "hot_fix":1
56   }
57 }
58
59
60 # 通过 bool 查询,
61 POST softwares/_search
62 {
63   "query": {
64     "bool": {
65       "filter": [
66         {
67           "match":{
68             "version.marjor":7
69           }
70         },
71         {
72           "match":{
```

```
73   "version.minor":2
74   }
75   }
76   ]
77   }
78   }
79   }
80
```

建模建议4：避免空值引起的聚合不准

```
1  # Not Null 解决聚合的问题
2  DELETE /scores
3  PUT /scores
4  {
5    "mappings": {
6      "properties": {
7        "score": {
8          "type": "float",
9          "null_value": 0
10       }
11     }
12   }
13 }
14
15 PUT /scores/_doc/1
16 {
17   "score": 100
18 }
19 PUT /scores/_doc/2
20 {
21   "score": null
22 }
23
24 POST /scores/_search
25 {
26   "size": 0,
27   "aggs": {
28     "avg": {
29       "avg": {
30         "field": "score"
```

```
31 }  
32 }  
33 }  
34 }
```

建模建议5: 为索引的Mapping加入Meta 信息

- Mappings设置非常重要, 需要从两个维度进行考虑
 - 功能: 搜索, 聚合, 排序
 - 性能: 存储的开销; 内存的开销; 搜索的性能
- Mappings设置是一个迭代的过程
 - 加入新的字段很容易 (必要时需要update_by_query)
 - 更新删除字段不允许(需要Reindex重建数据)
 - 最好能对Mappings 加入Meta 信息, 更好的进行版本管理
 - 可以考虑将Mapping文件上传git进行管理

```
1 PUT /my_index  
2 {  
3   "mappings": {  
4     "_meta": {  
5       "index_version_mapping": "1.1"  
6     }  
7   }  
8 }
```