Elasticsearch

基于开源的全文搜索引擎项目：Apache Lucene

由Java语言编写

**安装和运行：**

**交互3种：**

通用=>使用RESTful API进行交互，通过集群的9200端口(http.port)进行交互。

Java =>通过集群的9300(transport.tcp.port)端口进行交互，需要驱动版本与集群一致，跟随集群升级。

1 节点客户端(Node Client)：应用程序作为无数据节点加入集群，能够直接找到数据所在的节点，减少跳转。

2 传输客户端(Transport Client)：类似于应用连接数据库的方式，将请求发送到集群中。

RESTful API格式

curl –X<VERB> “<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY\_STRING>” -d ‘<BODY>’

VERB: http的方法，如GET, POST, PUT, DELETE, PATCH, HEAD

PROTOCOL: http或https

HOST: 集群的任意主机名

PORT: 服务的端口号，默认为9200

PATH: 服务API的名称（最终路径），如\_count

QUERY\_STRING: 查询参数，如?pretty

BODY: 查询的请求体

存储格式：JSON文档

文档元数据：

\_index 文档存放的位置（数据库），小写，不能以下划线开头，不能包含逗号

\_type 文档类型（表），不能以下划线或句号开头，不能包含逗号，长度<256字符

\_id 文档唯一标识，字符串，自行提供值（PUT）或由Elasticsearch生成（POST），自动生成的ID是URL-safe，基于Base64，长度20字符的GUID，由FlakedID模式生成。

\_version 文档版本，每次修改或删除时，版本号都会递增

**索引文档（Create）：**

如果\_id相同，则会覆盖文档

PUT /{index}/{type}/{id}

{

"field": "value",

...

}

使Elastisearch自动生成\_id

POST /website/blog/

{

"title": "My second blog entry",

"text": "Still trying this out...",

"date": "2014/01/01"

}

创建新文档（不覆盖）：使用op\_type=create或末端API为\_create，如果文档已存在，则返回Conflict 409状态码

PUT /website/blog/123?op\_type=create

{ ... }

或者

PUT /website/blog/123/\_create

{ ... }

**检出文档（Query）：**

GET /website/blog/123?pretty

GET /website/blog/123?\_source=title,text（返回文档的部分字段）

GET /website/blog/123/\_source（只返回source原文档）

**检查文档是否存在（Exist）：**

curl -i –X HEAD http://localhost:9200/website/blog/123

使用HEAD方法，如果存在则返回200 ok，不存在则返回404 not found

**更新文档（Update）：**

使用PUT方法，使用同个\_id进行创建，即可更新

PUT /website/blog/123

{

"title": "My first blog entry",

"text": "I am starting to get the hang of this...",

"date": "2014/01/02"

}

使用API

文档不可变，create一个\_id相同的文档或update一个文档时，会构建新的文档，同时标记旧文档为删除，后续新增更多文档时，后台会清理这些旧文档。

**删除文档（DELETE）**

DELETE /website/blog/123

返回结果，如果文档不存在则found为false。删除后\_version字段会增加，即使该文档已经被删除过，删除的文档不会立即从磁盘中删除，后续继续索引文档时会逐步清理。

{

"found" : true,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 3

}

**处理冲突（并发）**

Elasticsearch使用乐观并发控制，类似CompareAndSwap（CAS）操作

UPDATE、DELETE时指定\_version号，如果该版本与现存的文档不一致则操作失败，返回409Conflict。失败的请求由客户端进行控制，重试或返回。

如下

PUT /website/blog/1?version=1

{

"title": "My first blog entry",

"text": "Starting to get the hang of this..."

}

如果请求成功，则返回

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 2

"created": false

}

如果请求失败，则返回

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[blog][1]: version conflict, current [2], provided [1]",

"index": "website",

"shard": "3"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[blog][1]: version conflict, current [2], provided [1]",

"index": "website",

"shard": "3"

},

"status": 409

}

另一种方式，沿用外部数据库的字段值作为\_version，如存储在其他数据库的timestamp，当数据库更新时，使用该timestamp作为\_version更新Elasticsearch中的文档。

如下，新增\_version为5的文档：

PUT /website/blog/2?version=5&version\_type=external

{

"title": "My first external blog entry",

"text": "Starting to get the hang of this..."

}

更新该文档，预期\_version为10：

PUT /website/blog/2?version=10&version\_type=external

{

"title": "My first external blog entry",

"text": "This is a piece of cake..."

}

返回的结果类似于上面\_version号递增用法。

**分析，分析器**



分析过程：

将文本拆分为适合建立倒排索引的词条（Token）

将词条进行标准化，提高可搜索性

分析器：

1字符过滤器

文本将按顺序通过每个字符过滤器，功能上包含将阿拉伯语的数字转化为通用数字、删除HTML标记、将&转化为and等操作，即在分词前整理文本。

2分词器

按照分词器的功能将文本拆分为词条（Token）。

3Token过滤器

词条将按顺序通过每个Token过滤器，功能上改变词条（如小写化、提取词根）、删除词条（如a, the, and等无用词条）、增加词条（如添加同义词等），即标准化词条。

**1.1字符过滤器**

<https://www.elastic.co/guide/en/elasticsearch/reference/6.7/analysis-charfilters.html>

HTML Strip Char Filter（html\_strip）

过滤HTML标记

Mapping Char Filter（mapping）

自定义字符的映射关系

Pattern Replace Char Filter（pattern\_replace）

根据Java正则表达式匹配过滤字符

**2.1分词器分类：**

<https://www.elastic.co/guide/en/elasticsearch/reference/6.7/analysis-tokenizers.html>

标准分析器（standard）

根据Unicode联盟定义的单词边界拆分文本，去除标点，将词条小写。

简单分析器（simple）

通过非字母字符拆分文本，去除标点，小写化，并且不保留数字。

空格分析器（whitespace）

通过空格字符拆分文本，保留标点，不修改大小写。

Stop分词器（stop）

在简单分析器基础上，去除英文中对搜索无用的单词（如a, the, and等）。

Keyword分词器（keyword）

将整个输入作为词条，适合地址、邮编等情景下进行拆分。

Partten分词器（partten）

使用正则表达式匹配的方式将文本拆分为词条。

IK分词器（ik\_max\_word和ik\_smart）

插件，优秀的中文分词器，同时也适配对英文的分词，包含两种策略的分词器。

ik\_max\_word按照最细粒度拆分中文词语。

ik\_smart按照最粗粒度拆分中文词语。

3.1Token过滤器

<https://www.elastic.co/guide/en/elasticsearch/reference/6.7/analysis-tokenfilters.html>

Standard Token Filter

Do Nothing

ASCII Folding Token Filter

将非标准ASCII字符转换为等效ASCII项

Length Token Filter

删除过长或者过短的词条，max定义最长长度，min定义最短长度

Lowercase Token Filter

将所有词条转换为小写

Uppercase Token Filter

将所有词条转换为大写

Stemmer Token Filter

添加同义词词条，需要手动在配置文件中定义同义词

Reverse Token Filter

翻转输入的词条

Unique Token Filter

消除重复的词条

Trim Token Filter

消除词条周围的空格

Delimited Payload Token Filter

处理定义的分隔符

Keep Words Token Filter

只保留定义的词条

**全文搜索**

1 match\_all

匹配所有文档

{ "match\_all": {}}

2 match

标准匹配，在一个全文字段上进行查询，执行查询前将使用分析器拆分查询的文本

{ "match": { "tweet": "About Search" }}

3 multi\_match

多个字段上匹配

{

"multi\_match": {

"query": "full text search",

"fields": [ "title", "body" ]

}

}

4 range

区间查询

{

"range": {

"age": {

"gte": 20,

"lt": 30

}

}

}

5 term

精确查询，对于输入的文本不进行分析，因此用于数字、布尔、时间等not\_analyzed字段

{ "term": { "age": 26 }}

{ "term": { "date": "2014-09-01" }}

{ "term": { "public": true }}

{ "term": { "tag": "full\_text" }}

6 terms

同精确查询，允许多值匹配

{ "terms": { "tag": [ "search", "full\_text", "nosql" ] }}

7 exists, missing

查询字段有值或者无值的文档

{

"exists": {

"field": "title"

}

}

**组合多查询**

must

必须匹配的条件

must\_not

必须不匹配的条件

should

如果条件满足则增加\_score，不满足则无影响。用于修正文档的相关性

filter

必须匹配，但不影响\_score

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"range": { "date": { "gte": "2014-01-01" }}

}

}

}

constant\_score

对所有查询到的文档用一个常量赋予\_score，用于只需要执行filter的查询

{

"constant\_score": {

"filter": {

"term": { "category": "ebooks" }

}

}

}

本地IDF和全局IDF

词频/逆向文档频率 -> TF/IDF相关度算法

由于分片架构下的性能考虑，ES仅对当前分片中的搜索结果做相关度计算，得出本地IDF，不同分片上的文档分布不一定均匀，因此最终得到的结果不会按照严格的相关性排序。

在文档数不断增加后，分片上的文档逐渐趋向均匀，因此本地IDF会被均匀化。

需要严格按照全局IDF计算结果时，在查询中添加?search\_type=dfs\_query\_then\_fetch，使ES对每个分片的本地IDF再进行一次计算，得出全局IDF。

**相关度评分的理论**

1布尔模型

使用AND, OR, NOT条件对文档进行过滤，将所有不匹配的文档排除在搜索结果外

2词频（Term Frequency - TF）

词元在文本中出现的次数频率，频率越高，权重越大。

词 t 在文档 d 的词频（ tf ）是该词在文档中出现次数的平方根。

tf(t in d) = √n

3 逆向文档频率（Inversion Document Frequency - IDF）

词元在所有文档中出现的频率，频率越高，权重越小。

词 t 的逆向文档频率（ idf ）是：索引中文档数量除以所有包含该词的文档数，然后求其对数。

idf(t) = 1 + log ( numDocs / (docFreq + 1))

4 字段长度归一值（Field-Length Norm - norm）【重要】

文档中字段的长度，搜索的内容出现在较短的字段中，相关度较大。字段越短，权重越大。

字段长度归一值（ norm ）是字段中词数平方根的倒数。

norm(d) = 1 / √numTerms

以下三个因素：词频（term frequency）、逆向文档频率（inverse document frequency）和字段长度归一值（field-length norm）是在索引时计算并存储的。最后将它们结合在一起计算单个词在特定文档中的 权重 。

查询时输出相关度计算详情：

GET /\_search

{

"explain": true,

"query" : {

"term" : { "user" : "kimchy" }

}

}

5 向量空间模型（Vector Space Model）

用于比较多词查询，将文档和查询都以向量形式表示。对每个词都进行权重计算（默认使用TF/IDF算法，也有其他的如Okapi-BM25）