# 基础知识

## 基本概念

**索引（index）：**Es存储数据的地方。

**文档类型（type）：**类型，用于区分不同类型的数据。

**文档（document）**：索引和搜索时使用的主要数据载体，包含一个或多个存有数据的字段。

**字段（field）：**文档的一部分，包含名称和值两部分。

**集群（cluster**）和**节点（node）：**ES可以以单点或者集群方式运行，以一个整体对外提供search服务的所有节点组成cluster，组成这个cluster的各个节点叫做node。

**分片（shard）：**将一个完整的index分成若干部分存储在相同或不同的节点上，这些组成index的部分就叫做shard。

**副本（replica）**：副本是一个分片的精确复制，每个分片可以有零个或多个副本。

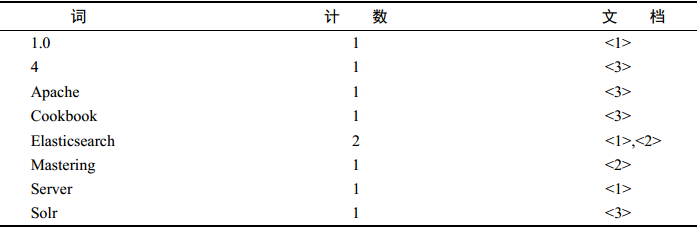
**倒排索引：**Es存储索引的结构

例如有以下文档：

Elasticsearch Server 1.0 (document 1)；

Mastering Elasticsearch (document 2)；

Apache Solr 4 Cookbook (document 3)

每一个词指向包含它的文档编号。这样就可以执行一种非常高效且快速的搜索，比如基于词的查询。此外，每个词有一个计数，告诉Lucene该词出现的频率。

**映射：**即mapping。

# RESTful API

## 请求类型

**GET：**用来获取请求对象的当前状态

**POST：**用来改变对象的当前状态

**DELETE：**销毁对象

**PUT：**创建一个对象

**HEAD：**用来获取对象的基础信息（不常用）

curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New version ofElasticsearch released!", content": "Version 1.0 released today!", "tags": ["announce","elasticsearch", "release"] }'

curl -XGET [http://localhost:9200/blog/article/1?pretty](http://localhost:9200/blog/article/_mapping?pretty)

curl -XDELETE <http://localhost:9200/blog/article>/1

## 更新文档

**过程：**

因为信息一旦在倒排索引中存储，就不能再被更改，所以更新文档要先删除旧的文档再索引新的文档。

## 用URI请求搜索

需要把请求发送到\_search端点，可以搜索一个或多个索引，也可以搜索一个或多个文档类型。

**搜索一个索引：**curl -XGET 'localhost:9200/books/\_search?pretty'

**搜索多个索引：**

curl -XGET 'localhost:9200/books,clients/\_search?pretty'

**指定类型：**curl -XGET 'localhost:9200/books/es/\_search?pretty'

**一个简单的搜索：**找到books索引中title字段包含elasticsearch一词的所有文档 curl -XGET 'localhost:9200/books/\_search?pretty&q=title:elasticsearch'

可以加上 explain=true 参数，查看额外的信息，分片及得分如何计算等。

可以加一个fields参数指定返回的字段，默认会返回所有字段，名为\_source

curl -XGET

'localhost:9200/blog/\_search?pretty&q=title:version&fields=title'

采用DSL查询如下:指定返回的字段

{

"fields" : [ "title", "year" ],

"query" : {

"query\_string" : { "query" : "title:crime" }

}

}

❑ 如果没有定义fields数组，它将用默认值，如果有就返回\_source字段；

❑ 如果使用\_source字段，并且请求一个没有存储的字段，那么这个字段将从\_source字段中提取

从性能的角度考虑，返回\_source字段比返回多个存储字段更好。

## 查询分析

请求\_analyze端点，查看title字段上短语建立的索引具体是什么

curl -XGET 'localhost:9200/books/\_analyze?field=title&pretty' -d 'Elasticsearch Server'

## 结果排序

默认按照得分排序，可以指定按某个字段排序 sort=publish\_time:desc

## 结果分页

从第11个开始返回5个文档 from=10&size=5，默认位置从0开始

采用DSL查询:

{

"from" : 9,

"size" : 20,

"query" : {

"query\_string" : { "query" : "title:crime" }

}

}

## 搜索类型

dfs\_query\_then\_fetch

dfs\_query\_and\_fetch

query\_then\_fetch

query\_and\_fetch

count

scan

## Lucene查询语法

操作符+告诉Lucene给定部分必须在文档中匹配。操作符-正相反，查询的这一部分不能出现在文档中。查询中既没有+又没有－操作符的部分将被视为可以匹配、但非强制性的查询。

+title:book -description:cat

可以用括号来组合多个词 title:(crime punishment)

可以用^指定权重 title:book^4

# 索引

## 分片和副本

Elasticsearch索引是由一个或多个分片组成的，每个分片包含了文档集的一部分,在创建索引是可以设置要建立的分片和副本的数量,如果按ES的默认配置,索引结束后会创建5个分片和5个分片副本,意味着将有将有10个Lucene索引分布在集群中。需要注意, 一旦创建好索引，更改分片数量的唯一途径就是创建另一个索引并重新索引数据,但是副本的数量可以在建完索引后再调整,那么分片和副本对我们索引和搜索有什么影响呢?

❑ 更多分片使索引能传送到更多服务器，意味着可以处理更多文件，而不会降低性能。

❑ 更多分片意味着获取特定文档所需的资源量会减少，因为相较于部署更少分片时，存储在单个分片中的文件数量更少。

❑ 更多分片意味着搜索索引时会面临更多问题，因为必须从更多分片中合并结果，使得查询的聚合阶段需要更多资源。

❑ 更多副本会增强集群系统的容错性，因为当原始分片不可用时，其副本将替代原始分片发挥作用。只拥有单个副本，集群可能在不丢失数据的情况下遗失分片。当有两个副本时，即使丢失了原始分片及其中一个副本，一切工作仍可以很好地持续下去。

❑ 更多副本意味着查询吞吐量将会增加，因为执行查询可以使用分片或分片的任一副本。

curl -XPUT http://localhost:9200/blog/ -d '{

"settings" : {

"number\_of\_shards" : 1,

"number\_of\_replicas" : 2

}

}'

## 创建索引

当我们创建一个文档时，如果索引不存在，Es会自动创建索引。有时候可能由于误操作发送了错误数据，如果Es自动创建索引，可能到时数据混乱，可以在Es配置文件elasticsearch.yml中加入action.auto\_create\_index: false

在我们的api中新建一个文档是按照以下过程：

1. curl -XPUT <http://localhost:9200/deal/>
2. 创建mapping
3. 索引数据

## 映射

"dynamic" : "false" 不会在现有索引中添加新字段

Es提供的字段类型：

❑ string：字符串；

❑ number：数字；

❑ date：日期；

❑ boolean：布尔型；

❑ binary：二进制。

这些类型的常用公共属性：

❑ index\_name：该属性定义将存储在索引中的字段名称

❑ index：可设置值为analyzed和no

如果设置为analyzed，该字段将被编入索引以供搜索。如果设置为no，将无法搜索该字段。默认值为analyzed。

在基于字符串的字段中，还有一个额外的选项not\_analyzed。此设置意味着字段将不经分析而编入索引，使用原始值被编入索引，在搜索的过程中必须全部匹配。

❑ store：这个属性的值可以是yes或no，指定了该字段的原始值是否被写入索引中。默认值设置为no，这意味着在结果中不能返回该字段（然而，如果你使用\_source字段，即使没有存储也可返回这个值），但是如果该值编入索引，仍可以基于它来搜索数据。

❑ boost：该属性的默认值是1。基本上，它定义了在文档中该字段的重要性。 boost的值  
越高，字段中值的重要性也越高。

❑ copy\_to：此属性指定一个字段，字段的所有值都将复制到该指定字段。

❑ include\_in\_all：此属性指定该字段是否应包括在\_all字段中。默认情况下，如果使用\_all字段，所有字段都会包括在其中

store 和 \_source 的区别

store的意思是，是否在\_source之外再独立存储一份，Es在索引数据的时候会保存一份源文档到\_source，如果某个字段设置为store=yes,那么会在\_source之外为这个字段独立进行存储，目的主要是针对内容比较多的字段，如果把所有字段都放到\_source返回，可能会占带宽，所以可以在\_source中exclude某些字段或禁掉返回\_source字段，而只返回这些字段。当然如果 \_source 不存储，并且 store 也为 no 这时候，数据就不会返回了，也不能高亮了。但是还是可以 match 的，前提是这个字段映射时 index 属性设置了 analyzed ，lucence 对这个字段建立了倒排索引。

❑ String 字符串类型：

"contents" : { "type" : "string", "store" : "no", "index" :"analyzed" }

❑ analyzer：该属性定义用于索引和搜索的分析器名称。它默认为全局定义的分析器名称。

❑ index\_analyzer：该属性定义了用于建立索引的分析器名称。

❑ search\_analyzer：该属性定义了的分析器，用于处理发送到特定字段的那部分查询字符串。

❑ ignore\_above：该属性定义字段中字符的最大值。当字段的长度高于指定值时，分析器会将其忽略。

❑ Number 数值类型：

❑ byte：定义字节值，例如1。

❑ short：定义短整型值，例如12。

❑ integer：定义整型值，例如134。

❑ long：定义长整型值，例如123456789。

❑ float：定义浮点值，例如12.23。

❑ double：定义双精度值，例如123.45。

❑ ignore\_malformed：此属性值可以设为true或false。默认值是false。若要忽略格式错误的值，则应设置属性值为true。

❑ date：日期：

❑ format：此属性指定日期的格式

"published" : { "type" : "date", "store" : "yes", "format" :"YYYY-mm-dd" }

{"name" : "Sample document","published" : "2012-12-22"}

❑ boolean：布尔型：

布尔值核心类型是专为索引布尔值（true或false）

"allowed" : { "type" : "boolean", "store": "yes" }

❑ binary：二进制：

是在索引中的二进制数据的Base64表示，可用来存储以二进制形式正常写入的数据，例如图像。基于此类型的字段在默认情况下只被存储，而不索引，因此只能提取，但无法对其执行搜索操作。二进制类型只支持index\_name属性。

基于binary字段的字段定义如下所示："image" : { "type" : "binary" }

## Analyzer 分析器

❑ standard：方便大多数欧洲语言的标准分析器

❑ simple：这个分析器基于非字母字符来分离所提供的值，并将其转换为小写形式

❑ whitespace：这个分析器基于空格字符来分离所提供的值

❑ pattern：这个分析器通过使用正则表达式灵活地分离文本

❑ language：这个分析器旨在特定的语言环境下工作

Elasticsearch 内置的分词器对中文不友好，会把中文分成单个字来进行全文检索，不能达到想要的结果，所以我们的代码中用的中文分词器 ik，ik可以自定义词库，根据我们系统的需要做一些定制，例如，设置一些词不做拆分，deal 搜索 ‘五折’就是配置的词库的词

## 批量索引

Elasticsearch可以合并多个请求至单个包中，而这些包可以单个请求的形式传送

❑在索引中增加或更换现有文档（index）；

❑从索引中移除文档（delete）；

❑当索引中不存在其他文档定义时，在索引中增加新文档（create）

{ "index": { "\_index": "addr", "\_type": "contact", "\_id": 1 }}

{ "name": "Fyodor Dostoevsky", "country": "RU" }

{ "create": { "\_index": "addr", "\_type": "contact", "\_id": 2 }}

{ "name": "Erich Maria Remarque", "country": "DE" }

{ "create": { "\_index": "addr", "\_type": "contact", "\_id": 2 }}

{ "name": "Joseph Heller", "country": "US" }

{ "delete": { "\_index": "addr", "\_type": "contact", "\_id": 4 }}

批量索引文件的大小存在限制，它被设定为100 MB，在Elasticsearch配置文件中可以通过  
http.max\_content\_length属性来改变。

将请求发送到Elasticsearch提供的\_bulk端点，形式可以是/\_bulk，也可以是/index\_  
name/\_bulk，甚至是/index\_name/type\_name/\_bulk

## 存储附加信息的字段介绍

标识符字段：第一个是\_uid字段，它是索引中文档的唯一标识符，由该文档的标识符和文档类型构成。第二个字段是\_id字段。此字段存储着索引时设置的实际标识符

\_type字段：类型标识

\_all字段：\_all字段存储其他字段中的数据以便于搜索，包含了索引中所有字段的所有数据，当然存储所有数据会造成数据很大，可以完全禁用\_all字段，或排除某些字段。

include\_in\_all：此属性指定该字段是否应包括在\_all字段中

\_source字段：在生成索引过程中存储发送到Elasticsearch的原始JSON文档。可以通过在\_source字段定义中添加includes或excludes属性来实现

"\_source" : {

"excludes" : [ "name" ]

}

也可以如下指定：

{

"partial\_fields" : {

"partial1" : {

"include" : [ "titl\*" ],

"exclude" : [ "chara\*" ]

}

}

\_index字段：可以帮助确定文档源自哪个索引

\_size字段：使我们能够自动索引\_source字段的原始大小

\_timestamp 字段：记录文档索引时间

\_ttl字段：表示time to live（生存时间），它允许定义文档的生命周期，周期结束之后文档会  
被自动删除。

## 段合并介绍

为什么需要段合并？占空间，影响搜索效率

Lucene库以及Elasticsearch中一旦数据被写入某些结构，就不再改变。由于段是无法改变的，因而有关删除的相关信息必须单独存储并动态应用到搜索过程中。这样做是为了从返回结果中去除已删除的文件。Elasticsearch支持文档更新，然而在底层，实际上是删除旧文档，再把更新内容的文档编入索引。随着时间的推移和持续索引数据，越来越多的段被创建。因此，搜索性能可能会降低，而且索引可能比原先大，因为它仍含有被删除的文件。

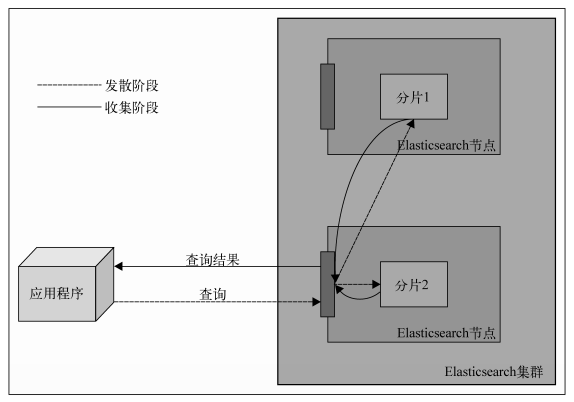
段合并的处理过程

底层的Lucene库获取若干段，并在这些段信息的基础上创建一个新的段。由此产生的段拥有所有存储在原始段中的文档，除了被标记为删除的那些之外。合并操作之后，源段将从磁盘上删除。这是因为段合并在CPU和I/O的使用方面代价是相当高的，关键是要适当地控制这个过程被调用的时机和频率。

Es有默认的段合并行为，我们也可以制定合并策略和合并调度器以及调节占用的I/O，可以通过配置改变，这里不做详细介绍了。

# 搜索

执行搜索请求的过程。



Es使用文档的唯一标识符来计算文档应该被放到哪个分片中。

收到查询请求的节点会把查询转发给保存了属于给定索引的分片的所有节点，并要求与查询匹配的文档的最少信息（默认情况下是标识符和得分）。这个过程称为发散阶段（scatter phase）。收到这些信息后，该聚合节点（收到客户端请求的节点）对结果排序，并发送第2个请求来获取结果列表所需的文档（除了标识符和得分以外的所有信息）。这个阶段称为收集阶段（gather phase）。这个阶段执行完毕后，结果返回到客户端。

## 搜索类型

❑ query\_then\_fetch：第一步，执行查询得到对文档进行排序和分级所需信息。这一步在所有的分片上执行。然后，只在相关分片上查询文档的实际内容。此查询类型返回结果的最大数量等于size参数的值。如果没有指定搜索类型，就默认使用这个类型

❑ query\_and\_fetch：这通常是最快也最简单的搜索类型实现。查询在所有分片上并行执行（当然，任意一个主分片，只查询一个副本），所有分片返回等于size值的结果数。返回文档的最大数量等于size的值乘以分片的数量

❑ dfs\_query\_and\_fetch：这个跟query\_and\_fetch类似，但相比query\_and\_fetch，它包含一个额外阶段，在初始查询中执行分布式词频的计算，以得到返回文件的更精确的得分，从而让查询结果更相关。

❑ dfs\_query\_then\_fetch：比query\_then\_fetch多一个额外的阶段，在初始查询中执行分布式词频的计算，以得到返回文件的更精确的得分，从而让查询结果更相关。

curl -XGET 'localhost:9200/library/book/\_search?pretty=true&search\_type=query\_and\_fetch' -d '{

"query" : {

"term" : { "title" : "crime" }

}

}'

## 基本查询

词条查询：它仅匹配在给定字段中含有该词条的文档，而且是确切的、未经分析的词条，注意，词条查询是未经分析的，因此需要提供跟索引文档中的词条完全匹配的词条。

{

"query" : {

"term" : {

"title" : "crime"

}

}

}

多词条查询：多词条查询可以用来匹配多个这未经分析的词条，minimum\_match属性设置为1意味着至少有1个词条应该匹配，这个设置类似should的设置，一会会讲到。

在tags字段中含有novel或book的文档：

{

"query" : {

"terms" : {

"tags" : [ "novel", "book" ],

"minimum\_match" : 1

}

}

}

match\_all查询: 它使我们能够匹配索引中的所有文件

{

"query" : {

"match\_all" : {}

}

}

常用词查询：

常用词查询，将查询分为两组。第一组包含重要的词，出现的频率较低。第二组包含较高频率的、不那么重要的词。先执行第一个查询， Elasticsearch从第一组的所有词中计算分数。这样，通常都很重要的低频词总是被列入考虑范围。然后， Elasticsearch对第二组中的词执行二次查询，但只为与第一个查询中匹配的文档计算得分。这样只计算了相关文档的得分，实现了更高的性能。

例如，“ crime and punishment”可以翻译成3个词查询，每一个都有性能上的成本（词越多，查询性能越低）。但“ and”这个词非常常见，对文档得分的影响非常低，另外两个词就是低频词，对文档得分影响比较大。

{

"query" : {

"common" : {

"title" : {

"query" : "crime and punishment",

"cutoff\_frequency" : 0.001

}

}

}

}

❑ query：这个参数定义了实际的查询内容。  
❑ cutoff\_frequency： 这个参数定义一个百分比（0.001表示0.1%）或一个绝对值（当此属性值>=1时）。这个值用来构建高、低频词组。此参数设置为0.001意味着频率<=0.1%的  
词将出现在低频词组中。

❑ low\_freq\_operator：这个参数可以设为or或and，默认是or。它用来指定为低频词组构建查询时用到的布尔运算符。如果希望所有的词都在文档中出现才认为是匹配，应该把它设置为and。

❑ high\_freq\_operator：这个参数可以设为or或and，默认是or。它用来指定为高频词组构建查询时用到的布尔运算符。如果希望所有的词都在文档中出现才认为是匹配，那么应该把它设置为and。

❑ minimum\_should\_match：不使用low\_freq\_operator和high\_freq\_operator参数的话，可以使用minimum\_should\_match参数。和其他查询一样，它允许指定匹配的文档中应该出现的查询词的最小个数。

❑ boost：这个参数定义了赋给文档得分的加权值。

❑ analyzer：这个参数定义了分析查询文本时用到的分析器名称。默认值为default

analyzer。

match查询:

match查询把query参数中的值拿出来，加以分析，然后构建相应的查询

{

"query" : {

"match" : {

"title" : "crime and punishment"

}

}

}

Match插叙两个常用参数：

❑operator：此参数可以接受or和and，控制用来连接创建的布尔条件的布尔运算符。默认值是or。如果希望查询中的所有条件都匹配，可以使用and运算符。

❑analyzer：这个参数定义了分析查询文本时用到的分析器的名字。默认值为default

analyzer

{

"query" : {

"match" : {

"title" : {

"query" : "crime and punishment",

"operator" : "and"

}

}

}

}

multi\_match查询：

multi\_match查询和match查询一样，不同的是它不是针对单个字段，而是可以通过fields参数针对多个字段查询，multi\_match和terms不一样，multi\_match是对多个字段，terms是一个字段的多个词条。

{

"query" : {

"multi\_match" : {

"query" : "crime punishment",

"fields" : [ "title", "otitle" ]

}

}

}

query\_string 查询:

query\_string查询支持全部的Apache Lucene查询语法

{

"query" : {

"query\_string" : {

"query" : "title:crime^10 +title:punishment -otitle:cat

+author:(+Fyodor +dostoevsky)",

"default\_field" : "title"

}

}

}

❑query：此参数指定查询文本。  
❑default\_field：此参数指定默认的查询字段，默认值由index.query.default\_  
field属性指定，默认为\_all。  
❑default\_operator：此参数指定默认的逻辑运算符（or或and），默认值是or。

query\_string也支持多个字段：

{

"query" : {

"query\_string" : {

"query" : "crime punishment",

"fields" : [ "title", "otitle" ]

}

}

}

simple\_query\_string 查询：

simple\_query\_string查询在解析错误时不会抛出异常。它丢弃查询无效的部分，执行其余部分

{

"query" : {

"simple\_query\_string" : {

"query" : "title:crime^10 +title:punishment -otitle:cat

+author:(+Fyodor +dostoevsky)",

"default\_operator" : "and"

}

}

}

范围查询：

范围查询使我们能够找到在某一字段值在某个范围里的文档，字段可以是数值型，也可以是基于字符串的。范围查询只能针对单个字段，查询参数应封装在字段名称中

❑gte：范围查询将匹配字段值大于或等于此参数值的文档。

❑gt：范围查询将匹配字段值大于此参数值的文档。

❑lte：范围查询将匹配字段值小于或等于此参数值的文档。

❑lt：范围查询将匹配字段值小于此参数值的文档

{

"query" : {

"range" : {

"year" : {

"gte" : 1700,

"lte" : 1900

}

}

}

}

## 复合查询

### 布尔查询

可以通过布尔查询来封装无限数量的查询。

❑should：被它封装的布尔查询可能被匹配，也可能不被匹配。被匹配的should节点数目由minimum\_should\_match参数控制。

❑must：被它封装的布尔查询必须被匹配，文档才会返回。

❑must\_not：被它封装的布尔查询必须不被匹配，文档才会返回。

结果文档的得分将由文档匹配的所有封装的查询得分总和计算得到

可以增加的参数：

❑boost：此参数指定了查询使用的加权值，默认为1.0。加权值越高，匹配文档的得分越高。

❑minimum\_should\_match：此参数的值描述了文档被视为匹配时，应该匹配的should子句的最少数量。

❑disable\_coord：此参数的默认值为false，允许启用或禁用分数因子的计算，该计算是基于文档包含的所有查询词条。如果得分不必太精确，但要查询快点，那么应该将它

设置为true。

{

"query": {

"bool": {

"must": {

"term": {

"title": "crime"

}

},

"should": {

"range": {

"year": {

"from": 1900,

"to": 2000

}

}

},

"must\_not": {

"term": {

"otitle": "nothing"

}

}

}

}

}

## 过滤器

应该尽可能使用过滤器。过滤器不影响评分，而得分计算让搜索变得复杂，而且需要CPU资源。另一方面，过滤是一种相对简单的操作。由于过滤应用在整个索引的内容上，过滤的结果独立于找到的文档，也独立于文档之间的关系。过滤器很容易被缓存，从而进一步提高过滤查询的整体性能。

如果可能，应该总是使用filtered查询，而不是post\_filter，因为filtered执行起来更快

{

"query": {

"match": {

"title": "Catch-22"

}

},

"post\_filter": {

"term": {

"year": 1961

}

}

}

{

"query": {

"filtered": {

"query": {

"match": {

"title": "Catch-22"

}

},

"filter": {

"term": {

"year": 1961

}

}

}

}

}

在第一种情况下，过滤器应用到查询所发现的所有文档上。第二种情况下，过滤发生在在运行查询之前，性能更好。如前所述，过滤器很快，所以filtered查询效率更高

## 数据排序

默认排序 "sort" : { "\_score" : "desc" }

选择用于排序的字段："sort" : [{ "title" : "asc" }]

## 使用建议器

### Context Suggester

{

"service": {

"properties": {

"name": {

"type" : "string"

},

"tag": {

"type" : "string"

},

"suggest\_field": {

"type": "completion",

"context": {

"color": {

"type": "category",

"path": "color\_field",

"default": ["red", "green", "blue"]

},

"location": {

"type": "geo",

"precision": "5m",

"neighbors": true,

"default": "u33"

}

}

}

}

}}

### Completion Suggester

{

"song" : {

"properties" : {

"name" : { "type" : "string" },

"suggest" : { "type" : "completion",

"index\_analyzer" : "simple",

"search\_analyzer" : "simple",

"payloads" : true

}

}

}

}