elasticsearch中文分詞組件安裝：

1.下載組件或者下載source code,如果是source code需要編譯

2.把編譯後的文件放置在新命名的/plugins/ik文件夾下，

3.重新啟動elasticsearch

4.在mapping中用ik\_max\_word或者ik\_smart模式進行分詞

5.取得分詞結果，使用post查詢192.168.56.101:9200/index/fulltext/\_search

{

“query”:{

“match”:{

“content”:”中國必勝”

}

},

“highlight”:{

“pre\_tags”:[“<tag1>”,”<tag2>”],

“post\_tags”:[“</tag1>”,”</tag2>”]

}

}

獲取節點信息：

GET http://192.168.56.101：9200/？pretty

計算文檔的數量

192.168.56.101:9200/\_count?pretty

{

“query”:{

“match\_all”:{}

}

}

查詢lastname為Smith的結果：

GET /megacorp/employee/\_search?q=last\_name:Smith

使用post查詢

{

“query”:{

“match”:{

“last\_name”:”Smith”

}

}

}

查找父姓為Smith，年齡大於32的那個僱員的信息：

{

“query”:{

“bool”:{

“must”:{

“match”:{

“last\_name”:”Smith”

}

},

“filter”:{

“range”:{

“age”:{

“gt”:30

}

}

}

}

}

}

全文檢索:

{

“query”:{

“match”:{

“about”:”rock climbing”

}

}

}

我们想执行这样一个查询，仅匹配同时包含 “rock” 和 “climbing” ，并且 二者以短语 “rock climbing” 的形式紧挨着的雇员记录。为此对 match 查询稍作调整，使用一个叫做 match\_phrase 的查询：

{

“query”:{

“match\_phrase”:{

“about”:”rock climbing”

}

}

}

高亮搜索结果：

{

“query”:{

“match”:{

“about”:”rock climbing”

}

},

“highlight”:{

“fields”:{

“about”:{}

}

}

}

持管理者对雇员目录做分析。 Elasticsearch 有一个功能叫聚合（aggregations），允许我们基于数据生成一些精细的分析结果。聚合与 SQL 中的 GROUP BY 类似但更强大。

举个例子，挖掘出雇员中最受欢迎的兴趣爱好

{

“aggs”:{

“all\_interests”:{

“terms”:{

“field”:”interests”

}

}

}

}

如果想知道叫 Smith 的雇员中最受欢迎的兴趣爱好，可以直接添加适当的查询来组合查询

{

“query”:{

“match”:{

“last\_name”:”Smith”

}

},

“aggs”:{

“all\_interests”:{

“terms”:{

“field”:”interests”

}

}

}

}

同樣愛好的人平均年齡是多少:

{

“aggs”:{

“all\_interests”:{

“terms”:{

“field”:”interests”

},

“aggs”:{

“avg\_age”:{

“field”:”age”

}

}

}

}

}

集群健康查詢：GET /\_cluster/health

在索引創建的時候已經確定了主分片數，但是副分片數可以隨時修改

PUT /blogs

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

調整index的副本數：

PUT /blogs/\_settings

{

"number\_of\_replicas" : 2

}

[JSON](http://en.wikipedia.org/wiki/Json) 是一种以人可读的文本表示对象的方法。 它已经变成 NoSQL 世界交换数据的事实标准。当一个对象被序列化成为 JSON，它被称为一个 JSON 文档 。

Elastcisearch 是分布式的 文档 存储。它能存储和检索复杂的数据结构--序列化成为JSON文档--以 实时 的方式。 换句话说，一旦一个文档被存储在 Elasticsearch 中，它就是可以被集群中的任意节点检索到。

索引文檔：

PUT /website/blog/123

{

"title": "My first blog entry",

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

"date": "2014/01/01"

}

不指定id索引文檔

POST /website/blog/

{

"title": "My second blog entry",

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

"date": "2014/01/01"

}

取回文檔：

GET /website/blog/123?pretty

GET /website/blog/123?\_source=title,text

GET /website/blog/123/\_source

ElasticSearch是分佈式的，當文檔創建、更新或刪除時，新版本的文檔必須複製到集群的其他節點。ElasticSearch也是異步和並發的，這意味著這些複製請求被並發的發送，並且到達目的地也許順序是亂的。Elasticsearch確保新的版本不會覆蓋舊的版本。

當我們之前討論index，Get和delete請求時，我們指出每個文檔都有一個version號，當文檔被修改時版本號遞增。我們可以利用\_version號達到這個目的。

新建一個index

PUT /website/blog/1/\_create

{

"title": "My first blog entry",

"text": "Just trying this out..."

}

檢索文檔：

GET /website/blog/1

更新指定的版本

PUT /website/blog/1?version=1 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"title": "My first blog entry",

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

}

通過外部版本號更新：

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

{

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

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

}

更新文本，指定文本號為10

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

{

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

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

}

文檔的部分更新:Update請求最簡單的一種形式是接收文檔的一部分作為doc的參數，它只是與現有的文檔合併，對象被合併到一起，覆蓋現有的字段，增加新的字段，

POST /website/blog/1/\_update

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

帶腳本更新：

POST /website/blog/1/\_update

{

“script”:{

“inline”:”ctx.\_source.tags.add(params.new\_tag)”,

“params”:{

“new\_tag”:”search”

}

}

}

POST /website/blog/1/\_update

{

“script”:{

“inline”:”ctx.op = ctx.\_source.views == params.count ? ‘delete’ : ’none’”,

“params”:{

“count”:1

}

}

}

獲取多個請求結果

GET /website/blog/\_mget

{

"docs" : [

{ "\_id" : 2 },

{ "\_type" : "pageviews", "\_id" : 1 }

]

}

/\_search

在所有的索引中搜索所有的类型

/gb/\_search

在 gb 索引中搜索所有的类型

/gb,us/\_search

在 gb 和 us 索引中搜索所有的文档

/g\*,u\*/\_search

在任何以 g 或者 u 开头的索引中搜索所有的类型

/gb/user/\_search

在 gb 索引中搜索 user 类型

/gb,us/user,tweet/\_search

在 gb 和 us 索引中搜索 user 和 tweet 类型

/\_all/user,tweet/\_search

在所有的索引中搜索 user 和 tweet 类型

設定延時時間

GET /\_search?timeout=10ms

如果每页展示 5 条结果，可以用下面方式请求得到 1 到 3 页的结果：

GET /\_search?size=5

GET /\_search?size=5&from=5

GET /\_search?size=5&from=10

es6.x數據集，通過\_bulk批量導入

post軟體把數據引入，把如下文本保存為.json格式。

{ "create": { "\_index": "user", "\_type": "doc", "\_id": "1" }}

{ "email" : "john@smith.com", "name" : "John Smith", "username" : "@john" }

{ "create": { "\_index": "user", "\_type": "doc", "\_id": "2" }}

{ "email" : "mary@jones.com", "name" : "Mary Jones", "username" : "@mary" }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "3" }}

{ "date" : "2014-09-13", "name" : "Mary Jones", "tweet" : "Elasticsearch means full text search has never been so easy", "user\_id" : 2 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "4" }}

{ "date" : "2014-09-14", "name" : "John Smith", "tweet" : "@mary it is not just text, it does everything", "user\_id" : 1 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "5" }}

{ "date" : "2014-09-15", "name" : "Mary Jones", "tweet" : "However did I manage before Elasticsearch?", "user\_id" : 2 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "6" }}

{ "date" : "2014-09-16", "name" : "John Smith", "tweet" : "The Elasticsearch API is really easy to use", "user\_id" : 1 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "7" }}

{ "date" : "2014-09-17", "name" : "Mary Jones", "tweet" : "The Query DSL is really powerful and flexible", "user\_id" : 2 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "8" }}

{ "date" : "2014-09-18", "name" : "John Smith", "user\_id" : 1 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "9" }}

{ "date" : "2014-09-19", "name" : "Mary Jones", "tweet" : "Geo-location aggregations are really cool", "user\_id" : 2 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "10" }}

{ "date" : "2014-09-20", "name" : "John Smith", "tweet" : "Elasticsearch surely is one of the hottest new NoSQL products", "user\_id" : 1 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "11" }}

{ "date" : "2014-09-21", "name" : "Mary Jones", "tweet" : "Elasticsearch is built for the cloud, easy to scale", "user\_id" : 2 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "12" }}

{ "date" : "2014-09-22", "name" : "John Smith", "tweet" : "Elasticsearch and I have left the honeymoon stage, and I still love her.", "user\_id" : 1 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "13" }}

{ "date" : "2014-09-23", "name" : "Mary Jones", "tweet" : "So yes, I am an Elasticsearch fanboy", "user\_id" : 2 }

{ "create": { "\_index": "tweet", "\_type": "doc", "\_id": "14" }}

{ "date" : "2014-09-24", "name" : "John Smith", "tweet" : "How many more cheesy tweets do I have to write?", "user\_id" : 1 }

空搜索

GET /\_search

查询字符串搜索非常适用于通过命令行做即席查询。例如，查询在 tweet 类型中 tweet 字段包含 elasticsearch 单词的所有文档：

GET /\_all/tweet/\_search?q=tweet:elasticsearch

下一个查询在 name 字段中包含 john 并且在 tweet 字段中包含 mary 的文档。实际的查询就是这样

+name:john +tweet:mary

但是查询字符串参数所需要的 *百分比编码* （译者注：URL编码）实际上更加难懂：

GET /\_search?q=%2Bname%3Ajohn+%2Btweet%3Amary

查詢推文，

GET /\_search?q=2014 # 12 results

GET /\_search?q=2014-09-15 # 12 results !

GET /\_search?q=date:2014-09-15 # 1 result

GET /\_search?q=date:2014 # 0 results !

測試分析器：

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

查看映射：

GET /gb/\_mapping/tweet

列出所有索引

\_cat/indices?v

使用post方法+bulk

http://192.168.56.101:9200/\_bulk + json語句

域最重要的属性是 type 。对于不是 string 的域，你一般只需要设置 type ：

{

"number\_of\_clicks": {

"type": "integer"

}

}

#### index[**编辑**](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/052_Mapping_Analysis/45_Mapping.asciidoc)

index 属性控制怎样索引字符串。它可以是下面三个值：

analyzed

首先分析字符串，然后索引它。换句话说，以全文索引这个域。

not\_analyzed

  索引这个域，所以它能够被搜索，但索引的是精确值。不会对它进行分析。

no

不索引这个域。这个域不会被搜索到。

string 域 index 属性默认是 analyzed 。如果我们想映射这个字段为一个精确值，我们需要设置它为 not\_analyzed ：

{

"tag": {

"type": "string",

"index": "not\_analyzed"

}

}

index 属性控制怎样索引字符串。它可以是下面三个值：

analyzed

首先分析字符串，然后索引它。换句话说，以全文索引这个域。

not\_analyzed

  索引这个域，所以它能够被搜索，但索引的是精确值。不会对它进行分析。

no

不索引这个域。这个域不会被搜索到。

string 域 index 属性默认是 analyzed 。如果我们想映射这个字段为一个精确值，我们需要设置它为 not\_analyzed ：

{

"tag": {

"type": "string",

"index": "not\_analyzed"

}

}

我们可以更新一个映射来添加一个新域，但不能将一个存在的域从 analyzed 改为 not\_analyzed 。

为了描述指定映射的两种方式，我们先删除 gd 索引：

DELETE /gb

然后创建一个新索引，指定 tweet 域使用 english 分析器：

PUT /gb https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"mappings": {

"tweet" : {

"properties" : {

"tweet" : {

"type" : "string",

"analyzer": "english"

},

"date" : {

"type" : "date"

},

"name" : {

"type" : "string"

},

"user\_id" : {

"type" : "long"

}

}

}

}

}

稍后，我们决定在 tweet 映射增加一个新的名为 tag 的 not\_analyzed 的文本域，使用 \_mapping ：

PUT /gb/\_mapping/tweet

{

"properties" : {

"tag" : {

"type" : "string",

"index": "not\_analyzed"

}

}

}

空查询将返回所有索引库（indices)中的所有文档：

GET /\_search

{} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

只用一个查询字符串，你就可以在一个、多个或者 \_all 索引库（indices）和一个、多个或者所有types中查询：

GET /index\_2014\*/type1,type2/\_search

{}

同时你可以使用 from 和 size 参数来分页：

GET /\_search

{

"from": 30,

"size": 10

}

空查詢在功能上等價於使用match\_all查詢，如下所示，

{

“query”:{

“match\_all”:{}

}

}

你可以使用match查詢tweet字段中包含elasticsearch的例子

post \_search

{

“query”:{

“match”:{

“tweet”:”elasticsearch”

}

}

}

查詢語句（query clauses）就像一些簡單的組合塊，這些組合塊可以彼此之間合併成更複雜的查詢，這些語句可以是如下形式，

葉子語句（leaf clauses）（就像match語句）用於將查詢字符串和一個字段（或者多個字段）對比

複合（compound）語句主要用於合併其他查詢。比如一個bool語句允許在你需要的時候組合其他語句，無論是must匹配，must\_not匹配還是should匹配，同時也可以包括不評分的過濾器

{

“query”:{

“bool”:{

“must”:{“match”:{“tweet”:”elasticsearch”}},

“must\_not”:{“match”:{“name”:”mary”}},

“should”:{“match”:{“tweet”:”full text”}},

“filter”:{“range”:{“age”:{“gt”:30}}}

}

}

}

一條複合語句可以合併任何其他查詢語句，包括複合語句，複合語句之間可以相互嵌套，以表達非常複雜的邏輯

例如，如下為找出信件正文包含business opportunity的星標郵件，或者在收件箱正文包含business opportunity的非垃圾郵件，

{

“bool”:{

“must”:{

“match”:{“mail”:”elasticsearch”}

},

“should”:[

{“match”:{“starred”:true}},

{“bool”:{

“must”:{“match”:{“folder”:”inbox”}},

“must\_not”:{“match”:{“spam”:true}}

}

}

],

“minimum\_should\_match”:1

}

}

过滤查询只是简单的检查或者排除，这使得计算特别快，至少一个过滤查询的结果是稀少的，并且经常使用不评分查询，结果会被缓存到内存中以便快速读取，所以有各种各样的手段来优化结果

评分查询不仅仅是要找出匹配的文档，还要计算每个文档的相关性，计算相关性使得他们比不评分查询费力的多，同时查询结果不缓存。

过滤的目标是减少那些需要通过评分查询进行检查的文档

通常的规则是，使用查询语句来进行全文搜索或者其他需要相关性得分的搜索，除此之外的情况都使用(filters).

最重要的查询

match\_all 查询简单的 匹配所有文档。在没有指定查询方式时，它是默认的查询：

{ "match\_all": {}}

无论你在任何字段上进行的是全文搜索还是精确查询，match 查询是你可用的标准查询。

如果你在一个全文字段上使用 match 查询，在执行查询前，它将用正确的分析器去分析查询字符串：

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

如果在一个精确值的字段上使用它， 例如数字、日期、布尔或者一个 not\_analyzed 字符串字段，那么它将会精确匹配给定的值：

{ "match": { "age": 26 }}

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

{ "match": { "public": true }}

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

multi\_match 查询可以在多个字段上执行相同的 match 查询：

{

"multi\_match": {

"query": "full text search",

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

}

}

range查詢，range查詢是找出那些落在指定區間內的數字或者時間，

{

“range”:{

“age”:{

“gt”:30

}

}

}

gt 大於，gte大於等於 lt 小於 lte 小於等於

term查詢用於精確值匹配，這些精確值可能是數字、時間、布爾或者那些not\_analyzed的字符串：

{“term”:{“age”:26}}

{“term”:{“date”:”2014-09-01”}}

{“term”:{“public”:true}}

{“term”:{“tag”:”full\_text”}}

terms查詢，

terms查詢和term查詢一樣，但它允許你指定多值進行匹配。如果這個字段包含指定值的任何一個值，那麼這個文檔滿足條件

{

“terms”:{

“tag”:[“search”,”full\_text”,”nosql”]

}

}

“exists”查詢和missing查詢:用於查找納西恩指定字段中有值(exists)或者無值（missing）的文檔。這與SQL中的IS\_NULL(Mmissing)和NOT IS\_NULL(exists)在本質上具有共性:

{

“exists”:{

“field”:”title”

}

}

range範圍查詢被包含在should語句中，

{

"bool":{

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

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

"should":[

{"match":{"tag":"starres"}},

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

]

}

}

如果沒有must語句，那麼至少需要能夠匹配其中一條should語句。但，如果存在至少一條must語句，則對should語句沒有要求。

增加帶過濾器(filtering)的查詢，

如果我們不想因為文檔的時間而影響得分，可以用filter語句來重寫前面的例子:

{

"bool":{

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

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

"should":[{"match":{"tag":"starred"}}],

"filter":{

"range":{

"date":{

"gte":"2014-01-01"

}

}

}

}

}

通過將range查詢移到filter語句中，我們可以將它轉成不評分的查詢，將不再影響文檔的相關性排名。由於它現在是一個不評分的查詢，可以使用各種對filter查詢有效的優化 手段來提升性能。

所有的查詢都可以借鑒這種方式。將查詢移轉到bool查詢的filter語句中，這樣它就自動的轉換成一個不評分的filter了。

如果你需要通過多個不同標準來過濾你的文檔，bool本身也可以被用作不評分的查詢，簡單的將它放置到filter語句中並在內部構件布爾邏輯。

{

"bool":{

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

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

"should":[{"tag":"starred"}],

"filter":{

"filter":{

"bool":{

"must":[

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

{"range":{"price":{"lte":29.99}}}

],

"must\_not":[{"term":{"category":"ebooks"}}]

}

}

}

}

}

儘管沒有bool查詢使用這麼頻繁，constant\_score查詢也是你工具箱中有用的查詢語句。它將一個不變的常量評分應用於所有匹配的文檔。他被經常用於執行一個filter而沒有其他查詢。

可以使用它來取代只有filter語句的bool查詢。在性能上是完全相同的，但對於查詢簡潔性和清晰度有很大的幫助

term查詢被放在constant\_score中，轉成不評分的filter。這種方式可以取代只有filter語句的bool查詢。

在elasticsearch中可以使用validate-query API可以用來驗證查詢是否合法

GET /gb/tweet/\_validate/query

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

为了找出 查询不合法的原因，可以将 explain 参数 加到查询字符串中：

GET /gb/tweet/\_validate/query?explain https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

对于合法查询，使用 explain 参数将返回可读的描述，这对准确理解 Elasticsearch 是如何解析你的 query 是非常有用的：

GET /\_validate/query?explain

{

"query": {

"match" : {

"tweet" : "really powerful"

}

}

}

elasticsearch離線安裝plugin

linux: sudo bin/elasticsearch-plugin install file:///path/to/plugin.zip

windows:

bin\elasticsearch-plugin install file:///C:/path/to/plugin.zip

To install a plugin from a HTTP URL:

sudo bin/elasticsearch-plugin install http://some.domain/path/to/plugin.zip

The plugin script will refuse to talk to an HTTPS URL with an untrusted certificate. To use a self-signed HTTPS cert, you will need to add the CA cert to a local Java truststore and pass the location to the script as follows:

sudo ES\_JAVA\_OPTS="-Djavax.net.ssl.trustStore=/path/to/trustStore.jks" bin/elasticsearch-plugin install https://host/plugin.zip

**es官方文档提供的ik\_max\_word和ik\_smart解释**

Elasticsearch 是一个分布式、可扩展、实时的搜索与数据分析引擎。

Elasticsearch 是一个开源的搜索引擎，建立在一个全文搜索引擎库 [Apache Lucene™](https://lucene.apache.org/core/) 基础之上。，

取而代之的提供一套简单一致的 RESTful API

默認端口: 9200

 Elasticsearch 内置的两个客户端：

节点客户端（Node client）

节点客户端作为一个非数据节点加入到本地集群中。换句话说，它本身不保存任何数据，但是它知道数据在集群中的哪个节点中，并且可以把请求转发到正确的节点。

传输客户端（Transport client）

轻量级的传输客户端可以将请求发送到远程集群。它本身不加入集群，但是它可以将请求转发到集群中的一个节点上。

两个 Java 客户端都是通过 *9300* 端口并使用 Elasticsearch 的原生 *传输* 协议和集群交互。集群中的节点通过端口 9300 彼此通信。如果这个端口没有打开，节点将无法形成一个集群。

Java 客户端作为节点必须和 Elasticsearch 有相同的 主要 版本；否则，它们之间将无法互相理解。

PUT /megacorp/employee/1

{

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

"about" : "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

GET /megacorp/employee/1

将 HTTP 命令由 PUT 改为 GET 可以用来检索文档，同样的，可以使用 DELETE 命令来删除文档，以及使用 HEAD 指令来检查文档是否存在。如果想更新已存在的文档，只需再次 PUT 。

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

GET /megacorp/employee/\_search

{

"query" : {

"bool": {

"must": {

"match" : {

"last\_name" : "smith" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"filter": {

"range" : {

"age" : { "gt" : 30 } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

全文搜索——一项传统数据库确实很难搞定的任务

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"about" : "rock climbing"

}

}

}

Elasticsearch 默认按照相关性得分排序，即每个文档跟查询的匹配程度。第一个最高得分的结果很明显：John Smith 的 about 属性清楚地写着 “rock climbing” 。

但为什么 Jane Smith 也作为结果返回了呢？原因是她的 about 属性里提到了 “rock” 。因为只有 “rock” 而没有 “climbing” ，所以她的相关性得分低于 John 的。

这是一个很好的案例，阐明了 Elasticsearch 如何 在 全文属性上搜索并返回相关性最强的结果。Elasticsearch中的 相关性 概念非常重要，也是完全区别于传统关系型数据库的一个概念，数据库中的一条记录要么匹配要么不匹配。

索引（名词）：

如前所述，一个 索引 类似于传统关系数据库中的一个 数据库 ，是一个存储关系型文档的地方。 索引 (index) 的复数词为 indices 或 indexes 。

索引（动词）：

索引一个文档 就是存储一个文档到一个 索引 （名词）中以便它可以被检索和查询到。这非常类似于 SQL 语句中的 INSERT 关键词，除了文档已存在时新文档会替换旧文档情况之外。

倒排索引：

关系型数据库通过增加一个 索引 比如一个 B树（B-tree）索引 到指定的列上，以便提升数据检索速度。Elasticsearch 和 Lucene 使用了一个叫做 倒排索引 的结构来达到相同的目的。

一个搜索默认返回十条结果。

GET /megacorp/employee/\_search

{

"query": {

"match": {

"last\_name": "smith"

}

},

"aggs": {

"all\_interests": {

"terms": {

"field": "interests"

}

}

}

}

* 分配文档到不同的容器 或 *分片* 中，文档可以储存在一个或多个节点中
* 按集群节点来均衡分配这些分片，从而对索引和搜索过程进行负载均衡
* 复制每个分片以支持数据冗余，从而防止硬件故障导致的数据丢失
* 将集群中任一节点的请求路由到存有相关数据的节点
* 集群扩容时无缝整合新节点，重新分配分片以便从离群节点恢复

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

*索引* —— 保存相关数据的地方。 索引实际上是指向一个或者多个物理 *分片* 的 *逻辑命名空间* 。

一个 *分片* 是一个底层的 *工作单元* ，它仅保存了 全部数据中的一部分。 在[分片内部机制](https://www.elastic.co/guide/cn/elasticsearch/guide/current/inside-a-shard.html)中，我们将详细介绍分片是如何工作的，而现在我们只需知道一个分片是一个 Lucene 的实例，以及它本身就是一个完整的搜索引擎。 我们的文档被存储和索引到分片内，但是应用程序是直接与索引而不是与分片进行交互。

Elasticsearch 是利用分片将数据分发到集群内各处的。分片是数据的容器，文档保存在分片内，分片又被分配到集群内的各个节点里。 当你的集群规模扩大或者缩小时， Elasticsearch 会自动的在各节点中迁移分片，使得数据仍然均匀分布在集群里。

一个分片可以是 *主* 分片或者 *副本* 分片。 索引内任意一个文档都归属于一个主分片，所以主分片的数目决定着索引能够保存的最大数据量。

一个副本分片只是一个主分片的拷贝。 副本分片作为硬件故障时保护数据不丢失的冗余备份，并为搜索和返回文档等读操作提供服务。

在索引建立的时候就已经确定了主分片数，但是副本分片数可以随时修改。

让我们在包含一个空节点的集群内创建名为 blogs 的索引。 索引在默认情况下会被分配5个主分片， 但是为了演示目的，我们将分配3个主分片和一份副本（每个主分片拥有一个副本分片）：

PUT /blogs

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

当你在同一台机器上启动了第二个节点时，只要它和第一个节点有同样的 cluster.name 配置，它就会自动发现集群并加入到其中。 但是在不同机器上启动节点的时候，为了加入到同一集群，你需要配置一个可连接到的单播主机列表。

Elasticsearch 默认被配置为使用单播发现，以防止节点无意中加入集群。只有在同一台机器上运行的节点才会自动组成集群。

使用单播，你可以为 Elasticsearch 提供一些它应该去尝试连接的节点列表。 当一个节点联系到单播列表中的成员时，它就会得到整个集群所有节点的状态，然后它会联系 master 节点，并加入集群。

这意味着你的单播列表不需要包含你的集群中的所有节点， 它只是需要足够的节点，当一个新节点联系上其中一个并且说上话就可以了。如果你使用 master 候选节点作为单播列表，你只要列出三个就可以了。 这个配置在 elasticsearch.yml 文件中：

discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]

所有新近被索引的文档都将会保存在主分片上，然后被并行的复制到对应的副本分片上。这就保证了我们既可以从主分片又可以从副本分片上获得文档。

分片是一个功能完整的搜索引擎，它拥有使用一个节点上的所有资源的能力。 我们这个拥有6个分片（3个主分片和3个副本分片）的索引可以最大扩容到6个节点，每个节点上存在一个分片，并且每个分片拥有所在节点的全部资源。

PUT /blogs/\_settings

{

"number\_of\_replicas" : 2

}

3个主分片和6个副本分片。 这意味着我们可以将集群扩容到9个节点，每个节点上一个分片。相比原来3个节点时，集群搜索性能可以提升 3 倍。

我们关闭的节点是一个主节点。而集群必须拥有一个主节点来保证正常工作，所以发生的第一件事情就是选举一个新的主节点： Node 2

为什么我们集群状态是 yellow 而不是 green 呢？ 虽然我们拥有所有的三个主分片，但是同时设置了每个主分片需要对应2份副本分片，而此时只存在一份副本分片。 所以集群不能为 green 的状态，

 一个对象仅仅是类似于 hash 、 hashmap 、字典或者关联数组的 JSON 对象，对象中也可以嵌套其他的对象。 对象可能包含了另外一些对象。在 Elasticsearch 中，术语 文档 有着特定的含义。它是指最顶层或者根对象, 这个根对象被序列化成 JSON 并存储到 Elasticsearch 中，指定了唯一 ID。

字段的名字可以是任何合法的字符串，但不可以包含时间段。

个必须的元数据元素如下：

\_index

文档在哪存放

\_type

文档表示的对象类别

\_id

文档唯一标识

\_index[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/030_Data/05_Document.asciidoc)

一个 *索引* 应该是因共同的特性被分组到一起的文档集合。

实际上，在 Elasticsearch 中，我们的数据是被存储和索引在 分片 中，而一个索引仅仅是逻辑上的命名空间， 这个命名空间由一个或者多个分片组合在一起。 然而，这是一个内部细节，我们的应用程序根本不应该关心分片，对于应用程序而言，只需知道文档位于一个 索引内。 Elasticsearch 会处理所有的细节。

所有需要我们做的就是选择一个索引名，这个名字必须小写，不能以下划线开头，不能包含逗号。

### \_type[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/030_Data/05_Document.asciidoc)

数据可能在索引中只是松散的组合在一起，但是通常明确定义一些数据中的子分区是很有用的。

这些文档共享一种相同的（或非常相似）的模式：他们有一个标题、描述、产品代码和价格。他们只是正好属于“产品”下的一些子类。

types （类型）的特性，它允许您在索引中对数据进行逻辑分区。不同 types 的文档可能有不同的字段，但最好能够非常相似。

### \_id[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/030_Data/05_Document.asciidoc)

ID 是一个字符串， 当它和 \_index 以及 \_type 组合就可以唯一确定 Elasticsearch 中的一个文档。 当你创建一个新的文档，要么提供自己的 \_id ，要么让 Elasticsearch 帮你生成。

如果你的数据没有自然的 ID， Elasticsearch 可以帮我们自动生成 ID 。 请求的结构调整为： 不再使用PUT 谓词(“使用这个 URL 存储这个文档”)， 而是使用 POST 谓词(“存储文档在这个 URL 命名空间下”)。

自动生成的 ID 是 URL-safe、 基于 Base64 编码且长度为20个字符的 GUID 字符串。 这些 GUID 字符串由可修改的 FlakeID 模式生成，这种模式允许多个节点并行生成唯一 ID ，且互相之间的冲突概率几乎为零。

GET 请求的响应体包括 {"found": true} ，这证实了文档已经被找到。 如果我们请求一个不存在的文档，我们仍旧会得到一个 JSON 响应体，但是 found 将会是 false 。 此外， HTTP 响应码将会是 404 Not Found ，而不是 200 OK 。

默认情况下， GET 请求 会返回整个文档，这个文档正如存储在 \_source 字段中的一样。但是也许你只对其中的 title 字段感兴趣。单个字段能用 \_source 参数请求得到，多个字段也能使用逗号分隔的列表来指定。

或者，如果你只想得到 \_source 字段，不需要任何元数据，你能使用 \_source 端点：

如果只想检查一个文档是否存在 --根本不想关心内容--那么用 HEAD 方法来代替 GET 方法。 HEAD 请求没有返回体，只返回一个 HTTP 请求报头：

curl -i -XHEAD http://localhost:9200/website/blog/123

如果文档存在， Elasticsearch 将返回一个 200 ok 的状态码：

HTTP/1.1 200 OK

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

若文档不存在， Elasticsearch 将返回一个 404 Not Found 的状态码：

curl -i -XHEAD http://localhost:9200/website/blog/124

HTTP/1.1 404 Not Found

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

当然，一个文档仅仅是在检查的时候不存在，并不意味着一毫秒之后它也不存在：也许同时正好另一个进程就创建了该文档。

在 Elasticsearch 中文档是 不可改变 的，不能修改它们。 相反，如果想要更新现有的文档，需要 重建索引或者进行替换， 我们可以使用相同的 index API 进行实现

在 Elasticsearch 中每个文档都有一个版本号。当每次对文档进行修改时（包括删除）， \_version 的值会递增。 在 [处理冲突](https://www.elastic.co/guide/cn/elasticsearch/guide/current/version-control.html) 中，我们讨论了怎样使用 \_version 号码确保你的应用程序中的一部分修改不会覆盖另一部分所做的修改。

更新一个文档的方法是检索并修改它，然后重新索引整个文档，

我们也介绍过文档是不可变的：他们不能被修改，只能被替换。 update API 必须遵循同样的规则。 从外部来看，我们在一个文档的某个位置进行部分更新。然而在内部， update API 简单使用与之前描述相同的 检索-修改-重建索引 的处理过程。 区别在于这个过程发生在分片内部，这样就避免了多次请求的网络开销。通过减少检索和重建索引步骤之间的时间，我们也减少了其他进程的变更带来冲突的可能性。

update 请求最简单的一种形式是接收文档的一部分作为 doc 的参数， 它只是与现有的文档进行合并。对象被合并到一起，覆盖现有的字段，增加新的字段。 例如，我们增加字段 tags 和 views 到我们的博客文章，

POST /website/blog/1/\_update

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

脚本可以在 update API中用来改变 \_source 的字段内容

默认的脚本语言 是 [Groovy](http://groovy.codehaus.org/)，一种快速表达的脚本语言，在语法上与 JavaScript 类似。 它在 Elasticsearch V1.3.0 版本首次引入并运行在 沙盒 中，然而 Groovy 脚本引擎存在漏洞， 允许攻击者通过构建 Groovy 脚本，在 Elasticsearch Java VM 运行时脱离沙盒并执行 shell 命令。

因此，在版本 v1.3.8 、 1.4.3 和 V1.5.0 及更高的版本中，它已经被默认禁用。 此外，您可以通过设置集群中的所有节点的 config/elasticsearch.yml 文件来禁用动态 Groovy 脚本：

script.groovy.sandbox.enabled: false

这将关闭 Groovy 沙盒，从而防止动态 Groovy 脚本作为请求的一部分被接受， 或者从特殊的 .scripts 索引中被检索。当然，你仍然可以使用存储在每个节点的 config/scripts/ 目录下的 Groovy 脚本。

如果你的架构和安全性不需要担心漏洞攻击，例如你的 Elasticsearch 终端仅暴露和提供给可信赖的应用， 当它是你的应用需要的特性时，你可以选择重新启用动态脚本。

如果我们尝试更新一个不存在的文档，那么更新操作将会失败。

在这样的情况下，我们可以使用 upsert 参数，指定如果文档不存在就应该先创建它：

POST /website/pageviews/1/\_update

{

"script" : "ctx.\_source.views+=1",

"upsert": {

"views": 1

}

}

在本节的介绍中，我们说明 检索 和 重建索引 步骤的间隔越小，变更冲突的机会越小。 但是它并不能完全消除冲突的可能性。 还是有可能在 update 设法重新索引之前，来自另一进程的请求修改了文档。

为了避免数据丢失， update API 在 检索 步骤时检索得到文档当前的 \_version 号，并传递版本号到 重建索引 步骤的 index 请求。 如果另一个进程修改了处于检索和重新索引步骤之间的文档，那么 \_version 号将不匹配，更新请求将会失败。

对于部分更新的很多使用场景，文档已经被改变也没有关系。 例如，如果两个进程都对页面访问量计数器进行递增操作，它们发生的先后顺序其实不太重要； 如果冲突发生了，我们唯一需要做的就是尝试再次更新。

这可以通过 设置参数 retry\_on\_conflict 来自动完成， 这个参数规定了失败之前 update 应该重试的次数，它的默认值为 0 。

POST /website/pageviews/1/\_update?retry\_on\_conflict=5 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"script" : "ctx.\_source.views+=1",

"upsert": {

"views": 0

}

}

|  |
| --- |
| 失败之前重试该更新5次。 |

在增量操作无关顺序的场景，例如递增计数器等这个方法十分有效，但是在其他情况下变更的顺序 *是* 非常重要的。 类似 [index API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/index-doc.html) ， update API 默认采用 *最终写入生效* 的方案，但它也接受一个 version 参数来允许你使用 [optimistic concurrency control](https://www.elastic.co/guide/cn/elasticsearch/guide/current/optimistic-concurrency-control.html) 指定想要更新文档的版本。

在 Elasticsearch 中每个文档都有一个版本号。当每次对文档进行修改时（包括删除）， \_version 的值会递增。 在 [处理冲突](https://www.elastic.co/guide/cn/elasticsearch/guide/current/version-control.html) 中，我们讨论了怎样使用 \_version 号码确保你的应用程序中的一部分修改不会覆盖另一部分所做的修改。

但实际上 Elasticsearch 按前述完全相同方式执行以下过程：

1. 从旧文档构建 JSON
2. 更改该 JSON
3. 删除旧文档
4. 索引一个新文档

唯一的区别在于, update API 仅仅通过一个客户端请求来实现这些步骤，而不需要单独的 get 和 index 请求。

当我们索引一个文档， 怎么确认我们正在创建一个完全新的文档，而不是覆盖现有的呢？

第一种方法使用 op\_type 查询 -字符串参数：

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

{ ... }

第二种方法是在 URL 末端使用 /\_create :

PUT /website/blog/123/\_create

{ ... }

如果创建新文档的请求成功执行，Elasticsearch 会返回元数据和一个 201 Created 的 HTTP 响应码。

另一方面，如果具有相同的 \_index 、 \_type 和 \_id 的文档已经存在，Elasticsearch 将会返回 409 Conflict 响应码，以及如下的错误信息：

{

"error": {

"root\_cause": [

{

"type": "document\_already\_exists\_exception",

"reason": "[blog][123]: document already exists",

"shard": "0",

"index": "website"

}

],

"type": "document\_already\_exists\_exception",

"reason": "[blog][123]: document already exists",

"shard": "0",

"index": "website"

},

"status": 409

删除文档 的语法和我们所知道的规则相同，只是 使用 DELETE 方法：

DELETE /website/blog/123

如果找到该文档，Elasticsearch 将要返回一个 200 ok 的 HTTP 响应码，和一个类似以下结构的响应体。注意，字段 \_version 值已经增加:

{

"found" : true,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 3

}

如果文档没有 找到，我们将得到 404 Not Found 的响应码和类似这样的响应体：

{

"found" : false,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 4

}

即使文档不存在（ Found 是 false ）， \_version 值仍然会增加。这是 Elasticsearch 内部记录本的一部分，用来确保这些改变在跨多节点时以正确的顺序执行。

正如已经在[更新整个文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/update-doc.html)中提到的，删除文档不会立即将文档从磁盘中删除，只是将文档标记为已删除状态。随着你不断的索引更多的数据，Elasticsearch 将会在后台清理标记为已删除的文档。

Elasticsearch 是分布式的。当文档创建、更新或删除时， 新版本的文档必须复制到集群中的其他节点。Elasticsearch 也是异步和并发的，这意味着这些复制请求被并行发送，并且到达目的地时也许 顺序是乱的。 Elasticsearch 需要一种方法确保文档的旧版本不会覆盖新的版本。

所有文档的更新或删除 API，都可以接受 version 参数，这允许你在代码中使用乐观的并发控制，

 Elasticsearch 不是检查当前 \_version 和请求中指定的版本号是否相同， 而是检查当前 \_version 是否 小于 指定的版本号。 如果请求成功，外部的版本号作为文档的新 \_version 进行存储。

外部版本号不仅在索引和删除请求是可以指定，而且在 创建 新文档时也可以指定。

例如，要创建一个新的具有外部版本号 5 的博客文章，我们可以按以下方法进行：

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

{

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

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

}

现在我们更新这个文档，指定一个新的 version 号是 10 ：

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

{

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

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

}

如果你要重新运行此请求时，它将会失败，并返回像我们之前看到的同样的冲突错误， 因为指定的外部版本号不大于 Elasticsearch 的当前版本号。

### 不要超过 32 GB！[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/510_Deployment/50_heap.asciidoc)

这里有另外一个原因不分配大内存给 Elasticsearch。事实上 ， JVM 在内存小于 32 GB 的时候会采用一个内存对象指针压缩技术。

在 Java 中，所有的对象都分配在堆上，并通过一个指针进行引用。 普通对象指针（OOP）指向这些对象，通常为 CPU 字长 的大小：32 位或 64 位，取决于你的处理器。指针引用的就是这个 OOP 值的字节位置。

对于 32 位的系统，意味着堆内存大小最大为 4 GB。对于 64 位的系统， 可以使用更大的内存，但是 64 位的指针意味着更大的浪费，因为你的指针本身大了。更糟糕的是， 更大的指针在主内存和各级缓存（例如 LLC，L1 等）之间移动数据的时候，会占用更多的带宽。

Java 使用一个叫作 [内存指针压缩（compressed oops）](https://wikis.oracle.com/display/HotSpotInternals/CompressedOops)的技术来解决这个问题。 它的指针不再表示对象在内存中的精确位置，而是表示 偏移量 。这意味着 32 位的指针可以引用 40 亿个 对象 ， 而不是 40 亿个字节。最终， 也就是说堆内存增长到 32 GB 的物理内存，也可以用 32 位的指针表示。

一旦你越过那个神奇的 ~32 GB 的边界，指针就会切回普通对象的指针。 每个对象的指针都变长了，就会使用更多的 CPU 内存带宽，也就是说你实际上失去了更多的内存。事实上，当内存到达 40–50 GB 的时候，有效内存才相当于使用内存对象指针压缩技术时候的 32 GB 内存。

这段描述的意思就是说：即便你有足够的内存，也尽量不要 超过 32 GB。因为它浪费了内存，降低了 CPU 的性能，还要让 GC 应对大内存。

即使有某个文档没有找到，上述请求的 HTTP 状态码仍然是 200 。事实上，即使请求 没有找到任何文档，它的状态码依然是 200 --因为 mget 请求本身已经成功执行。 为了确定某个文档查找是成功或者失败，你需要检查 found 标记。

通过批量索引典型文档，并不断增加批量大小进行尝试。 当性能开始下降，那么你的批量大小就太大了。一个好的办法是开始时将 1,000 到 5,000 个文档作为一个批次, 如果你的文档非常大，那么就减少批量的文档个数。

密切关注你的批量请求的物理大小往往非常有用，一千个 1KB 的文档是完全不同于一千个 1MB 文档所占的物理大小。 一个好的批量大小在开始处理后所占用的物理大小约为 5-15 MB。

我们可以发送请求到集群中的任一节点。 每个节点都有能力处理任意请求。 每个节点都知道集群中任一文档位置，所以可以直接将请求转发到需要的节点上。 在下面的例子中，将所有的请求发送到 Node 1 ，我们将其称为 协调节点(coordinating node) 。

当发送请求的时候， 为了扩展负载，更好的做法是轮询集群中所有的节点。

新建、索引和删除 请求都是 写 操作， 必须在主分片上面完成之后才能被复制到相关的副本分片

以下是使用单个 mget 请求取回多个文档所需的步骤顺序：

1. 客户端向 Node 1 发送 mget 请求。
2. Node 1 为每个分片构建多文档获取请求，然后并行转发这些请求到托管在每个所需的主分片或者副本分片的节点上。一旦收到所有答复， Node 1 构建响应并将其返回给客户端。

bulk API 按如下步骤顺序执行：

1. 客户端向 Node 1 发送 bulk 请求。
2. Node 1 为每个节点创建一个批量请求，并将这些请求并行转发到每个包含主分片的节点主机。
3. 主分片一个接一个按顺序执行每个操作。当每个操作成功时，主分片并行转发新文档（或删除）到副本分片，然后执行下一个操作。 一旦所有的副本分片报告所有操作成功，该节点将向协调节点报告成功，协调节点将这些响应收集整理并返回给客户端。

Elasticsearch可以直接读取被网络缓冲区接收的原始数据。 它使用换行符字符来识别和解析小的 action/metadata 行来决定哪个分片应该处理每个请求。

这些原始请求会被直接转发到正确的分片。没有冗余的数据复制，没有浪费的数据结构。整个请求尽可能在最小的内存中处理。

*映射（Mapping）*

描述数据在每个字段内如何存储

*分析（Analysis）*

全文是如何处理使之可以被搜索的

*领域特定查询语言（Query DSL）*

Elasticsearch 中强大灵活的查询语言

应当注意的是 timeout 不是停止执行查询，它仅仅是告知正在协调的节点返回到目前为止收集的结果并且关闭连接。在后台，其他的分片可能仍在执行查询即使是结果已经被发送了。

使用超时是因为 SLA(服务等级协议)对你是很重要的，而不是因为想去中止长时间运行的查询。

Elasticsearch 使用一种称为 倒排索引 的结构，它适用于快速的全文搜索。一个倒排索引由文档中所有不重复词的列表构成，对于其中每个词，有一个包含它的文档列表。

+ 前缀表明这个词必须存在

你只能搜索在索引中出现的词条，所以索引文本和查询字符串必须标准化为相同的格式。

分词和标准化的过程称为 分析

分析 包含下面的过程：

首先，将一块文本分成适合于倒排索引的独立的 词条 ，

之后，将这些词条统一化为标准格式以提高它们的“可搜索性”，或者 recall

字符过滤器

首先，字符串按顺序通过每个 字符过滤器 。他们的任务是在分词前整理字符串。一个字符过滤器可以用来去掉HTML，或者将 & 转化成 `and`。

分词器

其次，字符串被 分词器 分为单个的词条。一个简单的分词器遇到空格和标点的时候，可能会将文本拆分成词条。

Token 过滤器

最后，词条按顺序通过每个 token 过滤器 。这个过程可能会改变词条（例如，小写化 Quick ），删除词条（例如， 像 a`， `and`， `the 等无用词），或者增加词条（例如，像 jump 和 leap 这种同义词）。

Elasticsearch提供了开箱即用的字符过滤器、分词器和token 过滤器。 这些可以组合起来形成自定义的分析器以用于不同的目的。

当我们 索引 一个文档，它的全文域被分析成词条以用来创建倒排索引。 但是，当我们在全文域 搜索 的时候，我们需要将查询字符串通过 相同的分析过程 ，以保证我们搜索的词条格式与索引中的词条格式一致。

Elasticsearch 支持 如下简单域类型：

* 字符串: string
* 整数 : byte, short, integer, long
* 浮点数: float, double
* 布尔型: boolean
* 日期: date

除了我们提到的简单标量数据类型， JSON 还有 null 值，数组，和对象，这些 Elasticsearch 都是支持的。

很有可能，我们希望 tag 域 包含多个标签。我们可以以数组的形式索引标签：

对于数组，没有特殊的映射需求。任何域都可以包含0、1或者多个值，就像全文域分析得到多个词条。

这暗示 数组中所有的值必须是相同数据类型的 。你不能将日期和字符串混在一起。如果你通过索引数组来创建新的域，Elasticsearch 会用数组中第一个值的数据类型作为这个域的 类型 。

当你从 Elasticsearch 得到一个文档，每个数组的顺序和你当初索引文档时一样。你得到的 \_source 域，包含与你索引的一模一样的 JSON 文档。

但是，数组是以多值域 索引的—可以搜索，但是无序的。 在搜索的时候，你不能指定 “第一个” 或者 “最后一个”。 更确切的说，把数组想象成 装在袋子里的值 。

当然，数组可以为空。 这相当于存在零值。 事实上，在 Lucene 中是不能存储 null 值的，所以我们认为存在 null 值的域为空域。

下面三种域被认为是空的，它们将不会被索引：

"null\_value": null,

"empty\_array": [],

"array\_with\_null\_value": [ null ]

我们讨论的最后一个 JSON 原生数据类是 对象 -- 在其他语言中称为哈希，哈希 map，字典或者关联数组。

内部对象 经常用于 嵌入一个实体或对象到其它对象中。例如，与其在 tweet 文档中包含 user\_name 和 user\_id 域，我们也可以这样写：

{

"tweet": "Elasticsearch is very flexible",

"user": {

"id": "@johnsmith",

"gender": "male",

"age": 26,

"name": {

"full": "John Smith",

"first": "John",

"last": "Smith"

}

}

}

Lucene 不理解内部对象。 Lucene 文档是由一组键值对列表组成的。为了能让 Elasticsearch 有效地索引内部类，它把我们的文档转化成这样：

{

"tweet": [elasticsearch, flexible, very],

"user.id": [@johnsmith],

"user.gender": [male],

"user.age": [26],

"user.name.full": [john, smith],

"user.name.first": [john],

"user.name.last": [smith]

}

内部域 可以通过名称引用（例如， first ）。为了区分同名的两个域，我们可以使用全 路径 （例如， user.name.first ） 或 type 名加路径（ tweet.user.name.first ）。

在前面简单扁平的文档中，没有 user 和 user.name 域。Lucene 索引只有标量和简单值，没有复杂数据结构。

### 合并查询语句[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/054_Query_DSL/60_Query_DSL.asciidoc)

查询语句(Query clauses) 就像一些简单的组合块 ，这些组合块可以彼此之间合并组成更复杂的查询。这些语句可以是如下形式：

* 叶子语句（Leaf clauses） (就像 match 语句) 被用于将查询字符串和一个字段（或者多个字段）对比。
* 复合(Compound) 语句 主要用于 合并其它查询语句。 比如，一个 bool 语句 允许在你需要的时候组合其它语句，无论是 must 匹配、 must\_not 匹配还是 should 匹配，同时它可以包含不评分的过滤器（filters）：

{

"bool": {

"must": { "match": { "tweet": "elasticsearch" }},

"must\_not": { "match": { "name": "mary" }},

"should": { "match": { "tweet": "full text" }},

"filter": { "range": { "age" : { "gt" : 30 }} }

}

}

一条复合语句可以合并 任何 其它查询语句，包括复合语句，了解这一点是很重要的。这就意味着，复合语句之间可以互相嵌套，可以表达非常复杂的逻辑。

例如，以下查询是为了找出信件正文包含 business opportunity 的星标邮件，或者在收件箱正文包含business opportunity 的非垃圾邮件：

{

"bool": {

"must": { "match": { "email": "business opportunity" }},

"should": [

{ "match": { "starred": true }},

{ "bool": {

"must": { "match": { "folder": "inbox" }},

"must\_not": { "match": { "spam": true }}

}}

],

"minimum\_should\_match": 1

}

}

match\_all 查询简单的 匹配所有文档。在没有指定查询方式时，它是默认的查询：

{ "match\_all": {}}

它经常与 filter 结合使用-

无论你在任何字段上进行的是全文搜索还是精确查询，match 查询是你可用的标准查询。

如果你在一个全文字段上使用 match 查询，在执行查询前，它将用正确的分析器去分析查询字符串：

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

如果在一个精确值的字段上使用它， 例如数字、日期、布尔或者一个 not\_analyzed 字符串字段，那么它将会精确匹配给定的值：

{ "match": { "age": 26 }}

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

{ "match": { "public": true }}

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

对于精确值的查询，你可能需要使用 filter 语句来取代 query，因为 filter 将会被缓存。接下来，我们将看到一些关于 filter 的例子。

不像我们在 [轻量 搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-lite.html) 章节介绍的字符串查询（query-string search）， match 查询不使用类似 +user\_id:2 +tweet:search 的查询语法。它只是去查找给定的单词。这就意味着将查询字段暴露给你的用户是安全的；你需要控制那些允许被查询字段，不易于抛出语法异常。

multi\_match 查询可以在多个字段上执行相同的 match 查询：

{

"multi\_match": {

"query": "full text search",

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

}

}

range 查询找出那些落在指定区间内的数字或者时间：

{

"range": {

"age": {

"gte": 20,

"lt": 30

}

}

}

被允许的操作符如下：

gt

大于

gte

大于等于

lt

小于

lte

小于等于

term 查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/054_Query_DSL/70_Important_clauses.asciidoc)

term 查询被用于精确值 匹配，这些精确值可能是数字、时间、布尔或者那些 not\_analyzed 的字符串：

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

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

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

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

term 查询对于输入的文本不 分析 ，所以它将给定的值进行精确查询。

terms 查询和 term 查询一样，但它允许你指定多值进行匹配。如果这个字段包含了指定值中的任何一个值，那么这个文档满足条件：

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

和 term 查询一样，terms 查询对于输入的文本不分析。它查询那些精确匹配的值（包括在大小写、重音、空格等方面的差异）。

exists 查询和 missing 查询被用于查找那些指定字段中有值 (exists) 或无值 (missing) 的文档。这与SQL中的 IS\_NULL (missing) 和 NOT IS\_NULL (exists) 在本质上具有共性：

{

"exists": {

"field": "title"

}

}

这些查询经常用于某个字段有值的情况和某个字段缺值的情况。

你可以用 bool 查询来实现你的需求。这种查询将多查询组合在一起，成为用户自己想要的布尔查询。它接收以下参数：

must

文档 *必须* 匹配这些条件才能被包含进来。

must\_not

文档 *必须不* 匹配这些条件才能被包含进来。

should

如果满足这些语句中的任意语句，将增加 \_score ，否则，无任何影响。它们主要用于修正每个文档的相关性得分。

filter

*必须* 匹配，但它以不评分、过滤模式来进行。这些语句对评分没有贡献，只是根据过滤标准来排除或包含文档。

由于这是我们看到的第一个包含多个查询的查询，所以有必要讨论一下相关性得分是如何组合的。每一个子查询都独自地计算文档的相关性得分。一旦他们的得分被计算出来， bool 查询就将这些得分进行合并并且返回一个代表整个布尔操作的得分。

下面的查询用于查找 title 字段匹配 how to make millions 并且不被标识为 spam 的文档。那些被标识为 starred 或在2014之后的文档，将比另外那些文档拥有更高的排名。如果 \_两者\_ 都满足，那么它排名将更高：

{

"bool": {

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

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

"should": [

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

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

]

}

}

如果没有 must 语句，那么至少需要能够匹配其中的一条 should 语句。但，如果存在至少一条 must 语句，则对 should 语句的匹配没有要求。

果我们不想因为文档的时间而影响得分，可以用 filter 语句来重写前面的例子：

{

"bool": {

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

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

"should": [

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

],

"filter": {

"range": { "date": { "gte": "2014-01-01" }} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |
| --- |
| range 查询已经从 should 语句中移到 filter 语句 |

通过将 range 查询移到 filter 语句中，我们将它转成不评分的查询，将不再影响文档的相关性排名。由于它现在是一个不评分的查询，可以使用各种对 filter 查询有效的优化手段来提升性能。

所有查询都可以借鉴这种方式。将查询移到 bool 查询的 filter 语句中，这样它就自动的转成一个不评分的 filter 了。

如果你需要通过多个不同的标准来过滤你的文档，bool 查询本身也可以被用做不评分的查询。简单地将它放置到 filter 语句中并在内部构建布尔逻辑：

{

"bool": {

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

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

"should": [

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

],

"filter": {

"bool": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"must": [

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

{ "range": { "price": { "lte": 29.99 }}}

],

"must\_not": [

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

]

}

}

}

}

|  |
| --- |
| 将 bool 查询包裹在 filter 语句中，我们可以在过滤标准中增加布尔逻辑 |

通过混合布尔查询，我们可以在我们的查询请求中灵活地编写 scoring 和 filtering 查询逻辑。

constant\_score 查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/054_Query_DSL/75_Combining_queries_together.asciidoc)

尽管没有 bool 查询使用这么频繁，constant\_score 查询也是你工具箱里有用的查询工具。它将一个不变的常量评分应用于所有匹配的文档。它被经常用于你只需要执行一个 filter 而没有其它查询（例如，评分查询）的情况下。

可以使用它来取代只有 filter 语句的 bool 查询。在性能上是完全相同的，但对于提高查询简洁性和清晰度有很大帮助。

{

"constant\_score": {

"filter": {

"term": { "category": "ebooks" } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

term 查询被放置在 constant\_score 中，转成不评分的 filter。这种方式可以用来取代只有 filter 语句的 bool 查询。

validate-query API 可以用来验证查询是否合法。

GET /gb/tweet/\_validate/query

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

以上 validate 请求的应答告诉我们这个查询是不合法的：

{

"valid" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"failed" : 0

}

}

为了找出 查询不合法的原因，可以将 explain 参数 加到查询字符串中：

GET /gb/tweet/\_validate/query?explain https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

很明显，我们将查询类型(match)与字段名称 (tweet)搞混了：

{

"valid" : false,

"\_shards" : { ... },

"explanations" : [ {

"index" : "gb",

"valid" : false,

"error" : "org.elasticsearch.index.query.QueryParsingException:

[gb] No query registered for [tweet]"

} ]

}

对于合法查询，使用 explain 参数将返回可读的描述，这对准确理解 Elasticsearch 是如何解析你的 query 是非常有用的：

从 explanation 中可以看出，匹配 really powerful 的 match 查询被重写为两个针对 tweet 字段的 single-term 查询，一个single-term查询对应查询字符串分出来的一个term。

当然，对于索引 us ，这两个 term 分别是 really 和 powerful ，而对于索引 gb ，term 则分别是 realli 和 power 。之所以出现这个情况，是由于我们将索引 gb 中 tweet 字段的分析器修改为 english分析器。

在这个案例中，通过时间来对 tweets 进行排序是有意义的，最新的 tweets 排在最前。 我们可以使用 sort 参数进行实现：

|  |
| --- |
| score 不被计算, 因为它并没有用于排序。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_Sorting.html#CO27-3) | date 字段的值表示为自 epoch (January 1, 1970 00:00:00 UTC)以来的毫秒数，通过 sort 字段的值进行返回。 |

首先我们在每个结果中有一个新的名为 sort 的元素，它包含了我们用于排序的值。 在这个案例中，我们按照 date 进行排序，在内部被索引为 *自 epoch 以来的毫秒数* 。 long 类型数 1411516800000 等价于日期字符串 2014-09-24 00:00:00 UTC 。

其次 \_score 和 max\_score 字段都是 null 。 计算 \_score 的花销巨大，通常仅用于排序； 我们并不根据相关性排序，所以记录 \_score 是没有意义的。如果无论如何你都要计算 \_score ， 你可以将track\_scores 参数设置为 true 。

提示

一个简便方法是, 你可以 指定一个字段用来排序：

"sort": "number\_of\_children"

字段将会默认升序排序 ，而按照 \_score 的值进行降序排序。

一种情形是字段有多个值的排序， 需要记住这些值并没有固有的顺序；一个多值的字段仅仅是多个值的包装，这时应该选择哪个进行排序呢？

对于数字或日期，你可以将多值字段减为单值，这可以通过使用 min 、 max 、 avg 或是 sum 排序模式 。例如你可以按照每个 date 字段中的最早日期进行排序，通过以下方法：

"sort": {

"dates": {

"order": "asc",

"mode": "min"

}

}

每个文档都有相关性评分，用一个正浮点数字段 \_score 来表示 。 \_score 的评分越高，相关性越高。

查询语句会为每个文档生成一个 \_score 字段。评分的计算方式取决于查询类型 不同的查询语句用于不同的目的： fuzzy 查询会计算与关键词的拼写相似程度，terms 查询会计算 找到的内容与关键词组成部分匹配的百分比，但是通常我们说的 *relevance* 是我们用来计算全文本字段的值相对于全文本检索词相似程度的算法。

Elasticsearch 的相似度算法 被定义为检索词频率/反向文档频率， *TF/IDF* ，包括以下内容：

检索词频率

检索词在该字段出现的频率？出现频率越高，相关性也越高。 字段中出现过 5 次要比只出现过 1 次的相关性高。

反向文档频率

每个检索词在索引中出现的频率？频率越高，相关性越低。检索词出现在多数文档中会比出现在少数文档中的权重更低。

字段长度准则

字段的长度是多少？长度越长，相关性越低。 检索词出现在一个短的 title 要比同样的词出现在一个长的 content 字段权重更大。

单个查询可以联合使用 TF/IDF 和其他方式，比如短语查询中检索词的距离或模糊查询里的检索词相似度。

相关性并不只是全文本检索的专利。也适用于 yes|no 的子句，匹配的子句越多，相关性评分越高。

如果多条查询子句被合并为一条复合查询语句 ，比如 bool 查询，则每个查询子句计算得出的评分会被合并到总的相关性评分中。

JSON 形式的 explain 描述是难以阅读的， 但是转成 YAML 会好很多，只需要在参数中加上 format=yaml 。

当 explain 选项加到某一文档上时， explain api 会帮助你理解为何这个文档会被匹配，更重要的是，一个文档为何没有被匹配。

请求路径为 /index/type/id/\_explain ，如下所示：

GET /us/tweet/12/\_explain

{

"query" : {

"bool" : {

"filter" : { "term" : { "user\_id" : 2 }},

"must" : { "match" : { "tweet" : "honeymoon" }}

}

}

}

* 在搜索的时候，我们能通过搜索关键词快速得到结果集。
* 当排序的时候，我们需要倒排索引里面某个字段值的集合。换句话说，我们需要 转置 倒排索引。

转置 结构在其他系统中经常被称作 列存储 。实质上，它将所有单字段的值存储在单数据列中，这使得对其进行操作是十分高效的，例如排序。

在 Elasticsearch 中，Doc Values 就是一种列式存储结构，默认情况下每个字段的 Doc Values 都是激活的，Doc Values 是在索引时创建的，当字段索引时，Elasticsearch 为了能够快速检索，会把字段的值加入倒排索引中，同时它也会存储该字段的 `Doc Values`。

Elasticsearch 中的 Doc Values 常被应用到以下场景：

* 对一个字段进行排序
* 对一个字段进行聚合
* 某些过滤，比如地理位置过滤
* 某些与字段相关的脚本计算

因为文档值被序列化到磁盘，我们可以依靠操作系统的帮助来快速访问。当 working set 远小于节点的可用内存，系统会自动将所有的文档值保存在内存中，使得其读写十分高速； 当其远大于可用内存，操作系统会自动把 Doc Values 加载到系统的页缓存中，从而避免了 jvm 堆内存溢出异常。

字符过滤器

字符过滤器 用来 整理 一个尚未被分词的字符串。例如，如果我们的文本是HTML格式的，它会包含像 <p> 或者 <div> 这样的HTML标签，这些标签是我们不想索引的。我们可以使用 [html清除 字符过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-htmlstrip-charfilter.html)来移除掉所有的HTML标签，并且像把 &Aacute; 转换为相对应的Unicode字符 Á 这样，转换HTML实体。

一个分析器可能有0个或者多个字符过滤器。

分词器

一个分析器 *必须* 有一个唯一的分词器。 分词器把字符串分解成单个词条或者词汇单元。 标准 分析器里使用的 [标准 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-standard-tokenizer.html) 把一个字符串根据单词边界分解成单个词条，并且移除掉大部分的标点符号，然而还有其他不同行为的分词器存在。

例如， [关键词 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-keyword-tokenizer.html) 完整地输出 接收到的同样的字符串，并不做任何分词。 [空格 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-whitespace-tokenizer.html) 只根据空格分割文本 。 [正则 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-pattern-tokenizer.html) 根据匹配正则表达式来分割文本 。

词单元过滤器

经过分词，作为结果的 *词单元流* 会按照指定的顺序通过指定的词单元过滤器 。

词单元过滤器可以修改、添加或者移除词单元。我们已经提到过 [lowercase](http://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-lowercase-tokenizer.html)和 [stop 词过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-stop-tokenfilter.html) ，但是在 Elasticsearch 里面还有很多可供选择的词单元过滤器。 [词干过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-stemmer-tokenfilter.html) 把单词 遏制 为 词干。 [ascii\_folding 过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-asciifolding-tokenfilter.html)移除变音符，把一个像 "très" 这样的词转换为 "tres" 。 [ngram](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-ngram-tokenfilter.html) 和 [edge\_ngram 词单元过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-edgengram-tokenfilter.html) 可以产生 适合用于部分匹配或者自动补全的词单元。

类型 在 Elasticsearch 中表示一类相似的文档。 类型由 名称 —比如 user 或 blogpost —和 映射 组成。

映射, 就像数据库中的 schema ，描述了文档可能具有的字段或 属性 、 每个字段的数据类型—比如 string, integer 或 date —以及Lucene是如何索引和存储这些字段的。

类型可以很好的抽象划分相似但不相同的数据。但由于 Lucene 的处理方式，类型的使用有些限制。

在 Lucene 中，一个文档由一组简单的键值对组成。 每个字段都可以有多个值，但至少要有一个值。 类似的，一个字符串可以通过分析过程转化为多个值。Lucene 不关心这些值是字符串、数字或日期--所有的值都被当做 不透明字节 。

当我们在 Lucene 中索引一个文档时，每个字段的值都被添加到相关字段的倒排索引中。你也可以将未处理的原始数据 存储 起来，以便这些原始数据在之后也可以被检索到。

Elasticsearch 类型是 以 Lucene 处理文档的这个方式为基础来实现的。一个索引可以有多个类型，这些类型的文档可以存储在相同的索引中。

Lucene 没有文档类型的概念，每个文档的类型名被存储在一个叫 \_type 的元数据字段上。 当我们要检索某个类型的文档时, Elasticsearch 通过在 \_type 字段上使用过滤器限制只返回这个类型的文档。

Lucene 也没有映射的概念。 映射是 Elasticsearch 将复杂 JSON 文档 映射 成 Lucene 需要的扁平化数据的方式。

映射的最高一层被称为 *根对象* ，它可能包含下面几项：

* 一个 *properties* 节点，列出了文档中可能包含的每个字段的映射
* 各种元数据字段，它们都以一个下划线开头，例如 \_type 、 \_id 和 \_source
* 设置项，控制如何动态处理新的字段，例如 analyzer 、 dynamic\_date\_formats 和dynamic\_templates
* 其他设置，可以同时应用在根对象和其他 object 类型的字段上，例如 enabled 、 dynamic 和 include\_in\_all

我们已经在 [核心简单域类型](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html#core-fields) 和 [复杂核心域类型](https://www.elastic.co/guide/cn/elasticsearch/guide/current/complex-core-fields.html) 章节中介绍过文档字段和属性的三个 最重要的设置：

type

字段的数据类型，例如 string 或 date

index

字段是否应当被当成全文来搜索（ analyzed ），或被当成一个准确的值（ not\_analyzed ），还是完全不可被搜索（ no ）

analyzer

确定在索引和搜索时全文字段使用的 analyzer

默认地，Elasticsearch 在 \_source 字段存储代表文档体的JSON字符串。和所有被存储的字段一样， \_source 字段在被写入磁盘之前先会被压缩。

这个字段的存储几乎总是我们想要的，因为它意味着下面的这些：

* 搜索结果包括了整个可用的文档——不需要额外的从另一个的数据仓库来取文档。
* 如果没有 \_source 字段，部分 update 请求不会生效。
* 当你的映射改变时，你需要重新索引你的数据，有了\_source字段你可以直接从Elasticsearch这样做，而不必从另一个（通常是速度更慢的）数据仓库取回你的所有文档。
* 当你不需要看到整个文档时，单个字段可以从 \_source 字段提取和通过 get 或者 search 请求返回。
* 调试查询语句更加简单，因为你可以直接看到每个文档包括什么，而不是从一列id猜测它们的内容。
* 然而，存储 \_source 字段的确要使用磁盘空间。如果上面的原因对你来说没有一个是重要的，你可以用下面的映射禁用 \_source 字段：
* PUT /my\_index
* {
* "mappings": {
* "my\_type": {
* "\_source": {
* "enabled": false
* }
* }
* }
* }

在一个搜索请求里，你可以通过在请求体中指定 \_source 参数，来达到只获取特定的字段的效果：

GET /\_search

{

"query": { "match\_all": {}},

"\_source": [ "title", "created" ]

}

**我有一个 1 TB 内存的机器！**

这个 32 GB 的分割线是很重要的。那如果你的机器有很大的内存怎么办呢？ 一台有着 512–768 GB内存的服务器愈发常见。

首先，我们建议避免使用这样的高配机器（参考 [硬件](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hardware.html)）。

但是如果你已经有了这样的机器，你有三个可选项：

* 你主要做全文检索吗？考虑给 Elasticsearch 4 - 32 GB 的内存， 让 Lucene 通过操作系统文件缓存来利用余下的内存。那些内存都会用来缓存 segments，带来极速的全文检索。
* 你需要更多的排序和聚合？而且大部分的聚合计算是在数字、日期、地理点和 非分词 字符串上？你很幸运，你的聚合计算将在内存友好的 doc values 上完成！ 给 Elasticsearch 4 到 32 GB 的内存，其余部分为操作系统缓存内存中的 doc values。
* 你在对分词字符串做大量的排序和聚合（例如，标签或者 SigTerms，等等）不幸的是，这意味着你需要 fielddata，意味着你需要堆空间。考虑在单个机器上运行两个或多个节点，而不是拥有大量 RAM 的一个节点。仍然要坚持 50％ 原则。

假设你有个机器有 128 GB 的内存，你可以创建两个节点，每个节点内存分配不超过 32 GB。 也就是说不超过 64 GB 内存给 ES 的堆内存，剩下的超过 64 GB 的内存给 Lucene。

如果你选择这一种，你需要配置 cluster.routing.allocation.same\_shard.host: true 。 这会防止同一个分片（shard）的主副本存在同一个物理机上（因为如果存在一个机器上，副本的高可用性就没有了）。

记住，\_all 字段仅仅是一个 经过分词的 string 字段。它使用默认分词器来分析它的值，不管这个值原本所在字段指定的分词器。就像所有 string 字段，你可以配置 \_all 字段使用的分词器：

PUT /my\_index/my\_type/\_mapping

{

"my\_type": {

"\_all": { "analyzer": "whitespace" }

}

}

当 Elasticsearch 遇到文档中以前 未遇到的字段，它用 [dynamic mapping](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html) 来确定字段的数据类型并自动把新的字段添加到类型映射。

幸运的是可以用 dynamic 配置来控制这种行为 ，可接受的选项如下：

true

动态添加新的字段--缺省

false

忽略新的字段

strict

如果遇到新字段抛出异常

把 dynamic 设置为 false 一点儿也不会改变 \_source 的字段内容。 \_source 仍然包含被索引的整个JSON文档。只是新的字段不会被加到映射中也不可搜索。

日期检测可以通过在根对象上设置 date\_detection 为 false 来关闭：

PUT /my\_index

{

"mappings": {

"my\_type": {

"date\_detection": false

}

}

}

使用这个映射，字符串将始终作为 string 类型。如果你需要一个 date 字段，你必须手动添加。

Elasticsearch 判断字符串为日期的规则可以通过 [dynamic\_date\_formats setting](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/dynamic-field-mapping.html#date-detection) 来设置。

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{ "es": {

"match": "\*\_es", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"match\_mapping\_type": "string",

"mapping": {

"type": "string",

"analyzer": "spanish"

}

}},

{ "en": {

"match": "\*", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"match\_mapping\_type": "string",

"mapping": {

"type": "string",

"analyzer": "english"

}

}}

]

}}}

匹配字段名以 \_es 结尾的字段。

匹配其他所有字符串类型字段。

一个索引中的所有类型共享相同的字段和设置。 \_default\_ 映射更加方便地指定通用设置，而不是每次创建新类型时都要重复设置。 \_default\_ 映射是新类型的模板。在设置 \_default\_ 映射之后创建的所有类型都将应用这些缺省的设置，除非类型在自己的映射中明确覆盖这些设置。

例如，我们可以使用 \_default\_ 映射为所有的类型禁用 \_all 字段， 而只在 blog 类型启用：

PUT /my\_index

{

"mappings": {

"\_default\_": {

"\_all": { "enabled": false }

},

"blog": {

"\_all": { "enabled": true }

}

}

}

\_default\_ 映射也是一个指定索引 [dynamic templates](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-dynamic-mapping.html#dynamic-templates) 的好方法。

字段 \_source 的一个优点是在Elasticsearch中已经有整个文档。你不必从源数据中重建索引，而且那样通常比较慢。

为了有效的重新索引所有在旧的索引中的文档，用 [scroll](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scroll.html) 从旧的索引检索批量文档 ， 然后用 [bulk API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html) 把文档推送到新的索引中。

索引 *别名* 就像一个快捷方式或软连接，可以指向一个或多个索引，也可以给任何一个需要索引名的API来使用。别名 带给我们极大的灵活性，允许我们做下面这些：

* 在运行的集群中可以无缝的从一个索引切换到另一个索引
* 给多个索引分组 (例如， last\_three\_months)
* 给索引的一个子集创建 视图

一个别名可以指向多个索引，所以我们在添加别名到新索引的同时必须从旧的索引中删除它。这个操作需要原子化，这意味着我们需要使用 \_aliases 操作：

POST /\_aliases

{

"actions": [

{ "remove": { "index": "my\_index\_v1", "alias": "my\_index" }},

{ "add": { "index": "my\_index\_v2", "alias": "my\_index" }}

]

}

倒排索引包含一个有序列表，列表包含所有文档出现过的不重复个体，或称为 词项 ，对于每一个词项，包含了它所有曾出现过文档的列表。

 Elasticsearch 中的 文档 是有字段和值的结构化 JSON 文档。事实上，在 JSON 文档中， 每个被索引的字段都有自己的倒排索引。

倒排索引被写入磁盘后是 *不可改变* 的:它永远不会修改。 不变性有重要的价值：

* 不需要锁。如果你从来不更新索引，你就不需要担心多进程同时修改数据的问题。
* 一旦索引被读入内核的文件系统缓存，便会留在哪里，由于其不变性。只要文件系统缓存中还有足够的空间，那么大部分读请求会直接请求内存，而不会命中磁盘。这提供了很大的性能提升。
* 其它缓存(像filter缓存)，在索引的生命周期内始终有效。它们不需要在每次数据改变时被重建，因为数据不会变化。
* 写入单个大的倒排索引允许数据被压缩，减少磁盘 I/O 和 需要被缓存到内存的索引的使用量。

在 Elasticsearch 中，写入和打开一个新段的轻量的过程叫做 refresh 。 默认情况下每个分片会每秒自动刷新一次。这就是为什么我们说 Elasticsearch 是 近 实时搜索: 文档的变化并不是立即对搜索可见，但会在一秒之内变为可见。

这些行为可能会对新用户造成困惑: 他们索引了一个文档然后尝试搜索它，但却没有搜到。这个问题的解决办法是用 refresh API 执行一次手动刷新:

POST /\_refresh https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

POST /blogs/\_refresh https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

刷新（Refresh）所有的索引。

只刷新（Refresh） blogs 索引。

尽管刷新是比提交轻量很多的操作，它还是会有性能开销。 当写测试的时候， 手动刷新很有用，但是不要在生产环境下每次索引一个文档都去手动刷新。 相反，你的应用需要意识到 Elasticsearch 的近实时的性质，并接受它的不足。

并不是所有的情况都需要每秒刷新。可能你正在使用 Elasticsearch 索引大量的日志文件， 你可能想优化索引速度而不是近实时搜索， 可以通过设置 refresh\_interval ， 降低每个索引的刷新频率：

PUT /my\_logs

{

"settings": {

"refresh\_interval": "30s" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

每30秒刷新 my\_logs 索引。

refresh\_interval 可以在既存索引上进行动态更新。 在生产环境中，当你正在建立一个大的新索引时，可以先关闭自动刷新，待开始使用该索引时，再把它们调回来：

refresh\_interval 可以在既存索引上进行动态更新。 在生产环境中，当你正在建立一个大的新索引时，可以先关闭自动刷新，待开始使用该索引时，再把它们调回来：

PUT /my\_logs/\_settings

{ "refresh\_interval": -1 }

#關閉自動刷新

PUT /my\_logs/\_settings

{ "refresh\_interval": "1s" }

#設置自動刷新時間間隔

refresh\_interval 需要一个 持续时间 值， 例如 1s （1 秒） 或 2m （2 分钟）。 一个绝对值 1 表示的是 1毫秒 --无疑会使你的集群陷入瘫痪。

一次完整的提交会将段刷到磁盘，并写入一个包含所有段列表的提交点。Elasticsearch 在启动或重新打开一个索引的过程中使用这个提交点来判断哪些段隶属于当前分片。

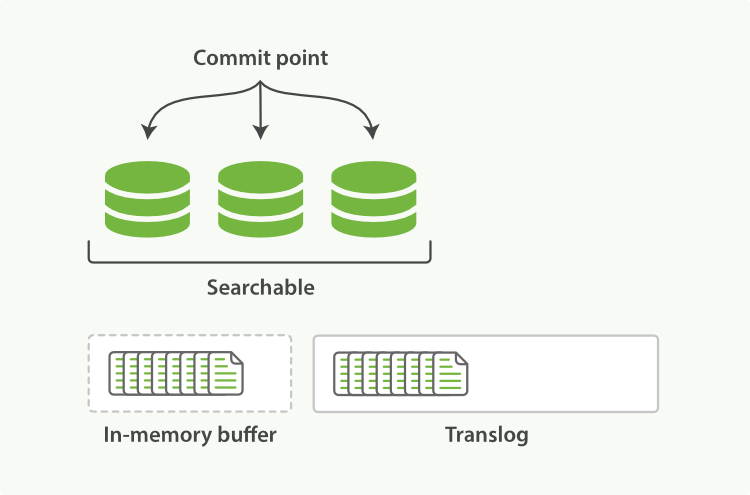
即使通过每秒刷新（refresh）实现了近实时搜索，我们仍然需要经常进行完整提交来确保能从失败中恢复。但在两次提交之间发生变化的文档怎么办？我们也不希望丢失掉这些数据。

Elasticsearch 增加了一个 translog ，或者叫事务日志，在每一次对 Elasticsearch 进行操作时均进行了日志记录。通过 translog ，整个流程看起来是下面这样：

Elasticsearch 增加了一个 translog ，或者叫事务日志，在每一次对 Elasticsearch 进行操作时均进行了日志记录。通过 translog ，整个流程看起来是下面这样：

一个文档被索引之后，就会被添加到内存缓冲区，并且 追加到了 translog ，正如 图 21 “新的文档被添加到内存缓冲区并且被追加到了事务日志” 描述的一样。

**图 21. 新的文档被添加到内存缓冲区并且被追加到了事务日志**



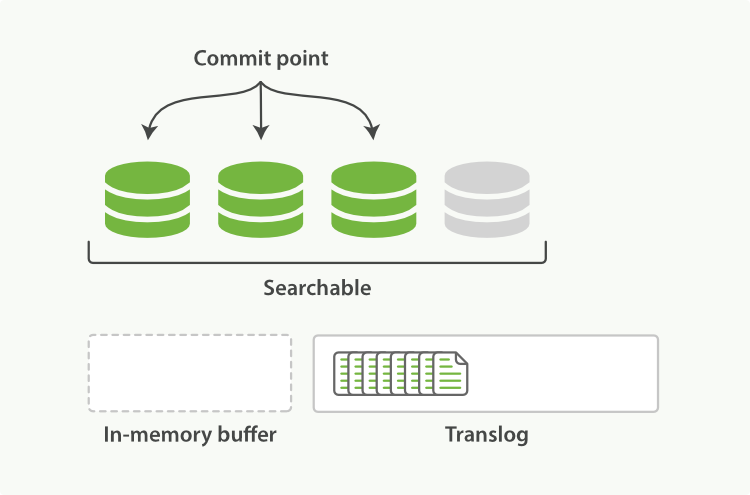
刷新（refresh）使分片处于 图 22 “刷新（refresh）完成后, 缓存被清空但是事务日志不会” 描述的状态，分片每秒被刷新（refresh）一次：

这些在内存缓冲区的文档被写入到一个新的段中，且没有进行 fsync 操作。

这个段被打开，使其可被搜索。

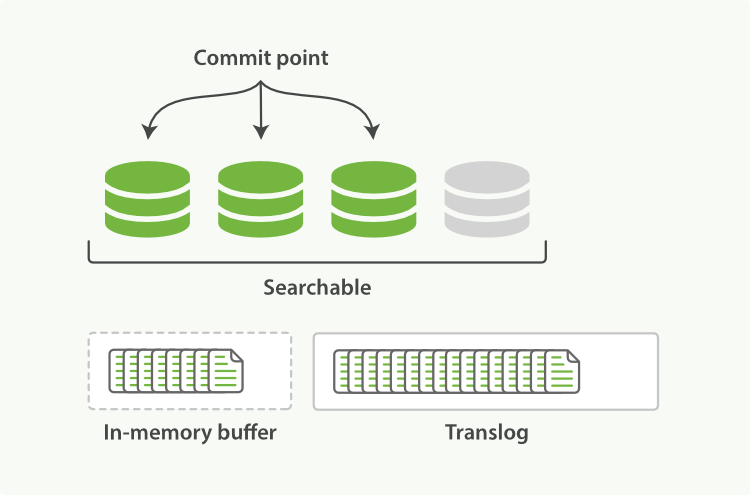
内存缓冲区被清空。

图 22. 刷新（refresh）完成后, 缓存被清空但是事务日志不会



这个进程继续工作，更多的文档被添加到内存缓冲区和追加到事务日志（见 图 23 “事务日志不断积累文档” ）。

图 23. 事务日志不断积累文档



每隔一段时间--例如 translog 变得越来越大--索引被刷新（flush）；一个新的 translog 被创建，并且一个全量提交被执行（见 图 24 “在刷新（flush）之后，段被全量提交，并且事务日志被清空” ）：

所有在内存缓冲区的文档都被写入一个新的段。

缓冲区被清空。

一个提交点被写入硬盘。

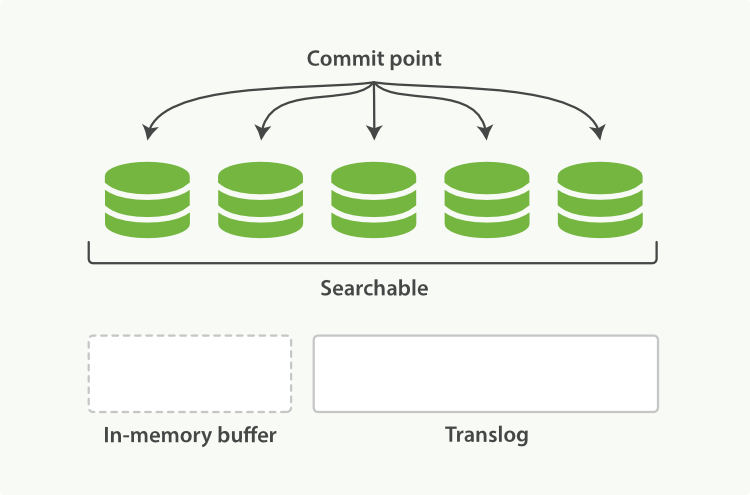
文件系统缓存通过 fsync 被刷新（flush）。

老的 translog 被删除。

translog 提供所有还没有被刷到磁盘的操作的一个持久化纪录。当 Elasticsearch 启动的时候， 它会从磁盘中使用最后一个提交点去恢复已知的段，并且会重放 translog 中所有在最后一次提交后发生的变更操作。

translog 也被用来提供实时 CRUD 。当你试着通过ID查询、更新、删除一个文档，它会在尝试从相应的段中检索之前， 首先检查 translog 任何最近的变更。这意味着它总是能够实时地获取到文档的最新版本。

图 24. 在刷新（flush）之后，段被全量提交，并且事务日志被清空



flush API编辑

这个执行一个提交并且截断 translog 的行为在 Elasticsearch 被称作一次 flush 。 分片每30分钟被自动刷新（flush），或者在 translog 太大的时候也会刷新。请查看 translog 文档 来设置，它可以用来 控制这些阈值：

flush API 可以 被用来执行一个手工的刷新（flush）:

POST /blogs/\_flush

POST /\_flush?wait\_for\_ongoing

你很少需要自己手动执行一个的 flush 操作；通常情况下，自动刷新就足够了。

这就是说，在重启节点或关闭索引之前执行 [flush](https://www.elastic.co/guide/cn/elasticsearch/guide/current/translog.html#flush-api) 有益于你的索引。当 Elasticsearch 尝试恢复或重新打开一个索引， 它需要重放 translog 中所有的操作，所以如果日志越短，恢复越快。

**Translog 有多安全?**

translog 的目的是保证操作不会丢失。这引出了这个问题： Translog 有多安全 ？

在文件被 fsync 到磁盘前，被写入的文件在重启之后就会丢失。默认 translog 是每 5 秒被 fsync 刷新到硬盘， 或者在每次写请求完成之后执行(e.g. index, delete, update, bulk)。这个过程在主分片和复制分片都会发生。最终， 基本上，这意味着在整个请求被 fsync 到主分片和复制分片的translog之前，你的客户端不会得到一个 200 OK 响应。

在每次请求后都执行一个 fsync 会带来一些性能损失，尽管实践表明这种损失相对较小（特别是bulk导入，它在一次请求中平摊了大量文档的开销）。

但是对于一些大容量的偶尔丢失几秒数据问题也并不严重的集群，使用异步的 fsync 还是比较有益的。比如，写入的数据被缓存到内存中，再每5秒执行一次 fsync 。

这个行为可以通过设置 durability 参数为 async 来启用：

PUT /my\_index/\_settings

{

"index.translog.durability": "async",

"index.translog.sync\_interval": "5s"

}

这个选项可以针对索引单独设置，并且可以动态进行修改。如果你决定使用异步 translog 的话，你需要 保证 在发生crash时，丢失掉 sync\_interval 时间段的数据也无所谓。请在决定前知晓这个特性。

如果你不确定这个行为的后果，最好是使用默认的参数（ "index.translog.durability": "request" ）来避免数据丢失。

段合并编辑

由于自动刷新流程每秒会创建一个新的段 ，这样会导致短时间内的段数量暴增。而段数目太多会带来较大的麻烦。 每一个段都会消耗文件句柄、内存和cpu运行周期。更重要的是，每个搜索请求都必须轮流检查每个段；所以段越多，搜索也就越慢。

Elasticsearch通过在后台进行段合并来解决这个问题。小的段被合并到大的段，然后这些大的段再被合并到更大的段。

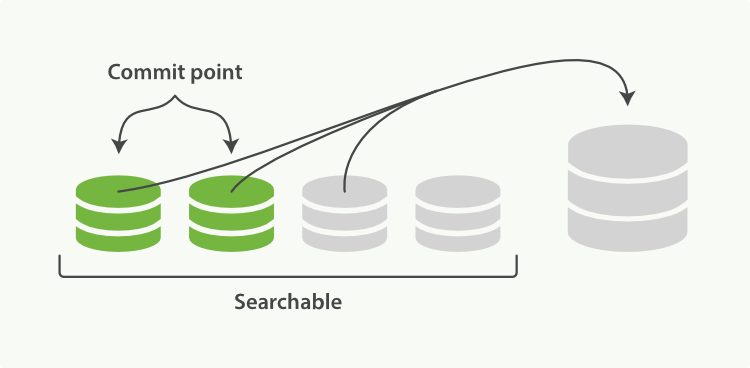
段合并的时候会将那些旧的已删除文档 从文件系统中清除。 被删除的文档（或被更新文档的旧版本）不会被拷贝到新的大段中。

启动段合并不需要你做任何事。进行索引和搜索时会自动进行。这个流程像在 图 25 “两个提交了的段和一个未提交的段正在被合并到一个更大的段” 中提到的一样工作：

1、 当索引的时候，刷新（refresh）操作会创建新的段并将段打开以供搜索使用。

2、 合并进程选择一小部分大小相似的段，并且在后台将它们合并到更大的段中。这并不会中断索引和搜索。

图 25. 两个提交了的段和一个未提交的段正在被合并到一个更大的段



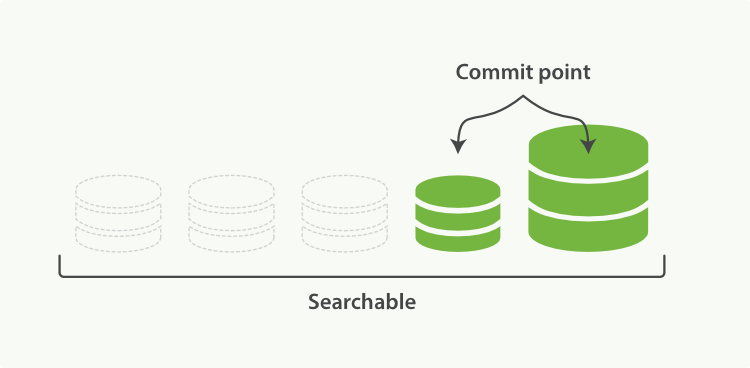
3、 图 26 “一旦合并结束，老的段被删除” 说明合并完成时的活动：

新的段被刷新（flush）到了磁盘。 \*\* 写入一个包含新段且排除旧的和较小的段的新提交点。

新的段被打开用来搜索。

老的段被删除。

图 26. 一旦合并结束，老的段被删除



optimize API

optimize API大可看做是 强制合并 API 。它会将一个分片强制合并到 max\_num\_segments 参数指定大小的段数目。 这样做的意图是减少段的数量（通常减少到一个），来提升搜索性能。

在特定情况下，使用 optimize API 颇有益处。例如在日志这种用例下，每天、每周、每月的日志被存储在一个索引中。 老的索引实质上是只读的；它们也并不太可能会发生变化。

在这种情况下，使用optimize优化老的索引，将每一个分片合并为一个单独的段就很有用了；这样既可以节省资源，也可以使搜索更加快速：

POST /logstash-2014-10/\_optimize?max\_num\_segments=1

#合併索引中的每個分片為一個單獨的段

# ElasticSearch在linux上安装部署

一.安装准备工作

安装参考文档:

ELK官网：https://www.elastic.co/

ELK官网文档：https://www.elastic.co/guide/index.html

ELK中文手册：https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html

ELK中文社区：https://elasticsearch.cn/

ELK-API :https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/transport-client.html

(1)、下载安装包

访问elasticSearch官网地址 https://www.elastic.co/

下载指定版本的安装包：elasticsearch-6.1.1.tar.gz

(2)、规划安装目录

     /export/servers/

(3)、上传安装包到指定目录

     通过FTP工具上传安装包到指定目录

(4)、解压安装包

tar -zxvf elasticsearch-6.1.1.tar.gz -C /export/servers

(5)、重命名安装目录

mv elasticsearch-6.1.1 elasticsearch

(6)、修改配置文件

进入到es安装目录下的config文件夹中，修改elasticsearch.yml 文件

扩展：

YML文件格式是YAML (YAML Aint Markup Language)编写的文件格式，YAML是一种直观的能够被电脑识别的的数据数据序列化格式，并且容易被人类阅读，容易和脚本语言交互的。

它的基本语法规则如下。

§ 大小写敏感

§ 使用缩进表示层级关系

§ 缩进时不允许使用Tab键，只允许使用空格。

§ 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可

# 表示注释，从这个字符一直到行尾，都会被解析器忽略

修改的主要内容：

#配置es的集群名称，默认是elasticsearch，es会自动发现在同一网段下的es，如果在同一网段下有多个集群，就可以用这个属性来区分不同的集群。

cluster.name: my-es

#节点名称

node.name: node-1

#设置索引数据的存储路径

path.data: /export/servers/data

#设置日志的存储路径

path.logs: /export/servers/logs

#设置当前的ip地址,通过指定相同网段的其他节点会加入该集群中

network.host: 192.168.216.121

#设置对外服务的http端口

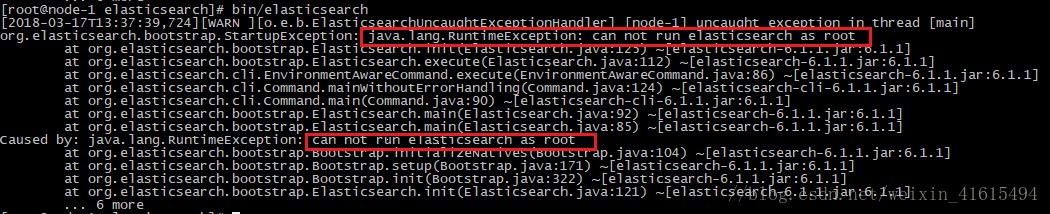
http.port: 9200

#设置集群中master节点的初始列表，可以通过这些节点来自动发现新加入集群的节点

discovery.zen.ping.unicast.hosts: ["node-1"]

二.解决启动时报错

(1)、在root用户下启动时报错



因为安全问题elasticsearch 不让用root用户直接运行，所以要创建新用户。

具体操作如下:

useradd es

passwd es

再输入两次密码(自定义)

mkdir -p /export/servers/data

mkdir -p /export/servers/logs

chown -R es:es /export/servers/elasticsearch

chown -R es:es /export/servers/data

chown -R es:es /export/servers/logs

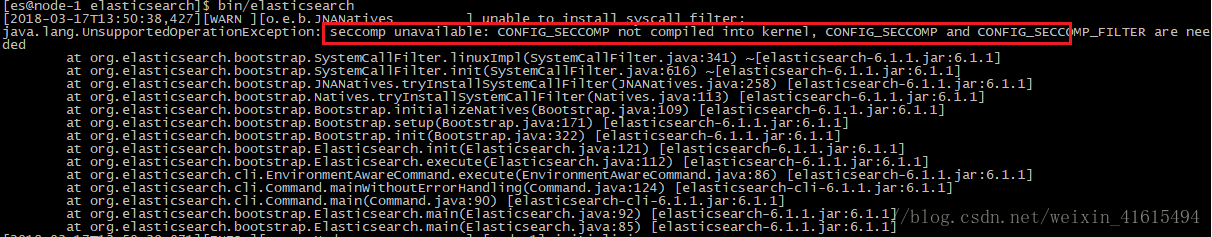
然后使用es用户启动

切换es用户命令：su es

cd /export/servers/elasticsearch

启动集群命令：bin/elasticsearch

(2)、在es用户下启动时报错

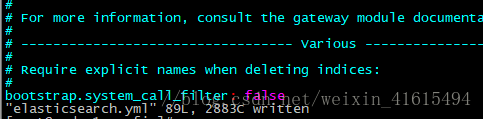


原因：Centos6不支持SecComp，而ES默认bootstrap.system\_call\_filter为true进行检测，所以导致检测失败，失败后直接导致ES不能启动。

详见 ：https://github.com/elastic/elasticsearch/issues/22899

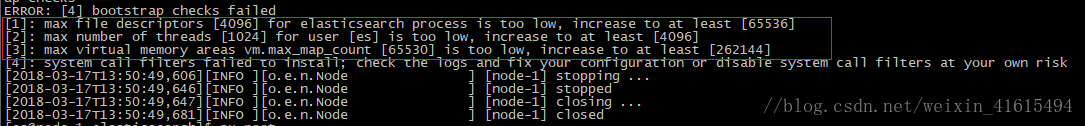
解决方案：

在elasticsearch.yml中新增配置bootstrap.system\_call\_filter，设为false。



bootstrap.system\_call\_filter: false

(3)、在es用户下启动继续报错



第一个问题的原因：

原因：无法创建本地文件问题,用户最大可创建文件数太小

解决方案：切换到root用户，编辑limits.conf配置文件， 添加类似如下内容：

vi /etc/security/limits.conf

添加如下内容: 注意\*不要去掉了

\* soft nofile 65536

\* hard nofile 131072

备注：\* 代表Linux所有用户名称（比如 hadoop）

需要保存、退出、重新登录才可生效。

第二个错误的原因：

原因：无法创建本地线程问题,用户最大可创建线程数太小

解决方案：切换到root用户，进入limits.d目录下，修改90-nproc.conf 配置文件。

vi /etc/security/limits.d/90-nproc.conf

找到如下内容：

\* soft nproc 1024

#修改为

\* soft nproc 4096

第三个错误的原因：

原因：最大虚拟内存太小

每次启动机器都手动执行下。

root用户执行命令：

执行命令：sysctl -w vm.max\_map\_count=262144

查看修改结果命令：sysctl -a|grep vm.max\_map\_count  看是否已经修改

永久性修改策略：

echo "vm.max\_map\_count=262144" >> /etc/sysctl.conf

三.测试

切换到es用户执行:

看到这个界面证明已经成功了,可以用浏览器访问192.168.216.121:9200查看,会下载一个json文件打开如下:

z

至此Elasticsearch就安装完成了,当然这只是一台机器,如果有需要还可以搭成集群.