# ELK学习笔记（来自龙果学院，内容非原创。）

## 分布式文档系统\_深度图解剖析document数据路由原理

### document路由到shard上是什么意思？

路由算法：shard = hash(routing) % number\_of\_primary\_shards

举个例子，一个index有3个primary shard，P0，P1，P2，每次增删改查一个document的时候，都会带过来一个routing number，默认就是这个document的\_id（可能是手动指定，也可能是自动生成），routing = \_id，假设\_id=1。会将这个routing值，传入一个hash函数中，产出一个routing值的hash值，假设hash(routing) = 21，然后将hash函数产出的值对这个index的primary shard的数量求余数，21 % 3 = 0，就决定了，这个document就放在P0上。

决定一个document在哪个shard上，最重要的一个值就是routing值，默认是\_id，也可以手动指定，相同的routing值，每次过来，从hash函数中，产出的hash值一定是相同的

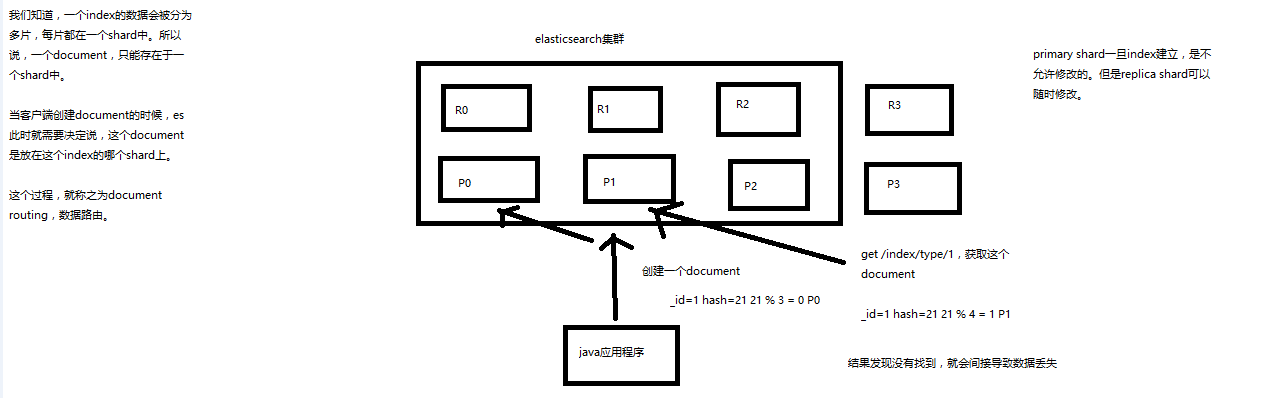
无论hash值是几，无论是什么数字，对number\_of\_primary\_shards求余数，结果一定是在0~number\_of\_primary\_shards-1之间这个范围内的。

### \_id or custom routing value

默认的routing就是\_id，也可以在发送请求的时候，手动指定一个routing value，比如说put /index/type/id?routing=user\_id

手动指定routing value是很有用的，可以保证说，某一类document一定被路由到一个shard上去，那么在后续进行应用级别的负载均衡，以及提升批量读取的性能的时候，是很有帮助的

### primary shard数量不可变的谜底



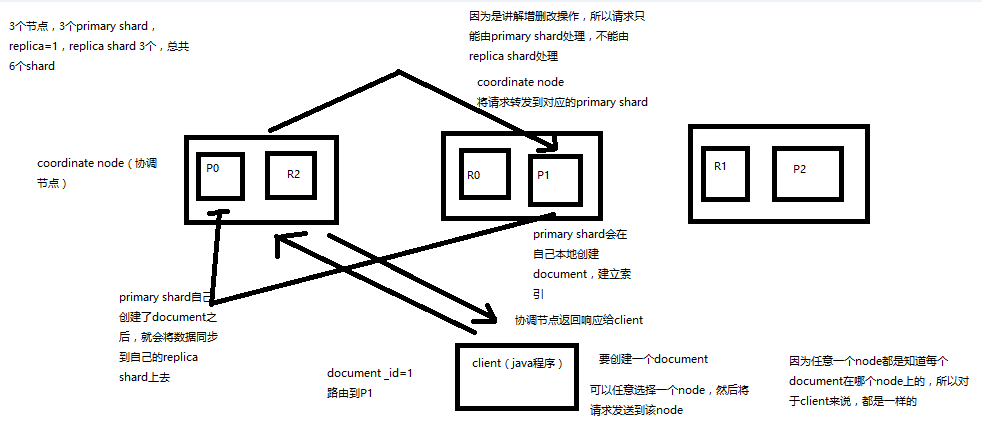
### 分布式文档系统\_document增删改内部原理图解揭秘

（1）客户端选择一个node发送请求过去，这个node就是coordinating node（协调节点）

（2） coordinating node，对document进行路由，将请求转发给对应的node（有primary shard）

（3）实际的node上的primary shard处理请求，然后将数据同步到replica node

（4）coordinating node，如果发现primary node和所有replica node都搞定之后，就返回响应结果给客户端。



## 分布式文档系统\_图解写一致性原理以及quorum机制深入剖析

（1）consistency，one（primary shard），all（all shard），quorum（default）

我们在发送任何一个增删改操作的时候，比如说put /index/type/id，都可以带上一个consistency参数，指明我们想要的写一致性是什么？

put /index/type/id?consistency=quorum

one：要求我们这个写操作，只要有一个primary shard是active活跃可用的，就可以执行

all：要求我们这个写操作，必须所有的primary shard和replica shard都是活跃的，才可以执行这个写操作

quorum：默认的值，要求所有的shard中，必须是大部分的shard都是活跃的，可用的，才可以执行这个写操作

（2）quorum机制，写之前必须确保大多数shard都可用，int( (primary + number\_of\_replicas) / 2 ) + 1，当number\_of\_replicas>1时才生效

quroum = int( (primary + number\_of\_replicas) / 2 ) + 1

举个例子，3个primary shard，number\_of\_replicas=1，总共有3 + 3 \* 1 = 6个shard

quorum = int( (3 + 1) / 2 ) + 1 = 3

所以，要求6个shard中至少有3个shard是active状态的，才可以执行这个写操作

（3）如果节点数少于quorum数量，可能导致quorum不齐全，进而导致无法执行任何写操作

3个primary shard，replica=1，要求至少3个shard是active，3个shard按照之前学习的shard&replica机制，必须在不同的节点上，如果说只有2台机器的话，是不是有可能出现说，3个shard都没法分配齐全，此时就可能会出现写操作无法执行的情况

es提供了一种特殊的处理场景，就是说当number\_of\_replicas>1时才生效，因为假如说，你就一个primary shard，replica=1，此时就2个shard

(1 + 1 / 2) + 1 = 2，要求必须有2个shard是活跃的，但是可能就1个node，此时就1个shard是活跃的，如果你不特殊处理的话，导致我们的单节点集群就无法工作

（4）quorum不齐全时，wait，默认1分钟，timeout，100，30s

等待期间，期望活跃的shard数量可以增加，最后实在不行，就会timeout

我们其实可以在写操作的时候，加一个timeout参数，比如说put /index/type/id?timeout=30，这个就是说自己去设定quorum不齐全的时候，es的timeout时长，可以缩短，也可以增长

## 分布式文档系统\_document查询内部原理图解揭秘

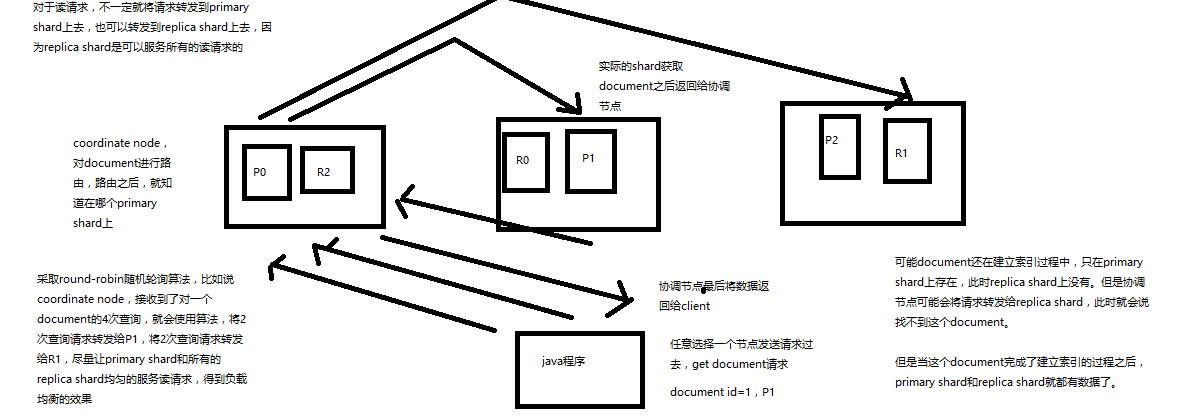
1、客户端发送请求到任意一个node，成为coordinate node

2、coordinate node对document进行路由，将请求转发到对应的node，此时会使用round-robin随机轮询算法，在primary shard以及其所有replica中随机选择一个，让读请求负载均衡

3、接收请求的node返回document给coordinate node

4、coordinate node返回document给客户端

5、特殊情况：document如果还在建立索引过程中，可能只有primary shard有，任何一个replica shard都没有，此时可能会导致无法读取到document，但是document完成索引建立之后，primary shard和replica shard就都有了



## 分布式文档系统\_bulk api的奇特json格式与底层性能优化关系大揭秘

bulk api奇特的json格式

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\n

良好的json数组格式

[{

"action": {

},

"data": { }

}]

1、bulk中的每个操作都可能要转发到不同的node的shard去执行

2、如果采用比较良好的json数组格式

允许任意的换行，整个可读性非常棒，读起来很爽，es拿到那种标准格式的json串以后，要按照下述流程去进行处理

（1）将json数组解析为JSONArray对象，这个时候，整个数据，就会在内存中出现一份一模一样的拷贝，一份数据是json文本，一份数据是JSONArray对象

（2）解析json数组里的每个json，对每个请求中的document进行路由

（3）为路由到同一个shard上的多个请求，创建一个请求数组

（4）将这个请求数组序列化

（5）将序列化后的请求数组发送到对应的节点上去

3、耗费更多内存，更多的jvm gc开销

我们之前提到过bulk size最佳大小的那个问题，一般建议说在几千条那样，然后大小在10MB左右，所以说，可怕的事情来了。假设说现在100个bulk请求发送到了一个节点上去，然后每个请求是10MB，100个请求，就是1000MB = 1GB，然后每个请求的json都copy一份为jsonarray对象，此时内存中的占用就会翻倍，就会占用2GB的内存，甚至还不止。因为弄成jsonarray之后，还可能会多搞一些其他的数据结构，2GB+的内存占用。

占用更多的内存可能就会积压其他请求的内存使用量，比如说最重要的搜索请求，分析请求，等等，此时就可能会导致其他请求的性能急速下降

另外的话，占用内存更多，就会导致java虚拟机的垃圾回收次数更多，跟频繁，每次要回收的垃圾对象更多，耗费的时间更多，导致es的java虚拟机停止工作线程的时间更多

4、现在的奇特格式

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\n

（1）不用将其转换为json对象，不会出现内存中的相同数据的拷贝，直接按照换行符切割json

（2）对每两个一组的json，读取meta，进行document路由

（3）直接将对应的json发送到node上去

5、最大的优势在于，不需要将json数组解析为一个JSONArray对象，形成一份大数据的拷贝，浪费内存空间，尽可能地保证性能

## 初识搜索引擎\_search结果深入解析（search timeout机制揭秘）

1、我们如果发出一个搜索请求的话，会拿到一堆搜索结果，这个搜索结果里的各种数据，都代表了什么含义？

GET /\_search

{

"took": 6,

"timed\_out": false,

"\_shards": {

"total": 6,

"successful": 6,

"failed": 0

},

"hits": {

"total": 10,

"max\_score": 1,

"hits": [

{

"\_index": ".kibana",

"\_type": "config",

"\_id": "5.2.0",

"\_score": 1,

"\_source": {

"buildNum": 14695

}

}

]

}

}

took：整个搜索请求花费了多少毫秒

hits.total：本次搜索，返回了几条结果

hits.max\_score：本次搜索的所有结果中，最大的相关度分数是多少，每一条document对于search的相关度，越相关，\_score分数越大，排位越靠前

hits.hits：默认查询前10条数据，完整数据，\_score降序排序

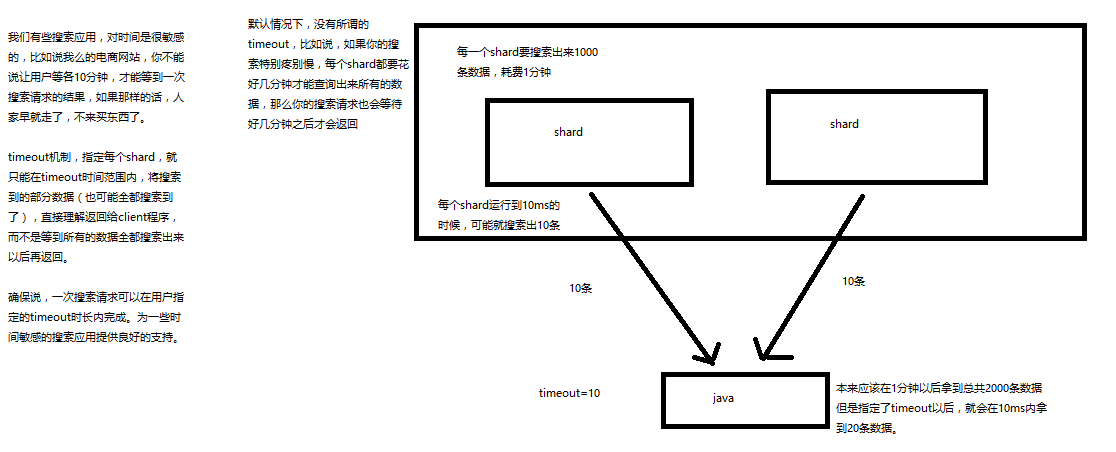
shards：shards fail的条件（primary和replica全部挂掉），不影响其他shard。默认情况下来说，一个搜索请求，会打到一个index的所有primary shard上去，当然了，每个primary shard都可能会有一个或多个replic shard，所以请求也可以到primary shard的其中一个replica shard上去。

2、搜索的timeout机制，底层的原理，画图讲解

timeout：默认无timeout，latency平衡completeness，手动指定timeout，timeout查询执行机制

timeout=10ms，timeout=1s，timeout=1m

GET /\_search?timeout=10m



## 初识搜索引擎\_multi-index&multi-type搜索模式解析以及搜索原理初步图解

1、multi-index和multi-type搜索模式

如何一次性搜索多个index和多个type下的数据

/\_search：所有索引，所有type下的所有数据都搜索出来

/index1/\_search：指定一个index，搜索其下所有type的数据

/index1,index2/\_search：同时搜索两个index下的数据

/\*1,\*2/\_search：按照通配符去匹配多个索引

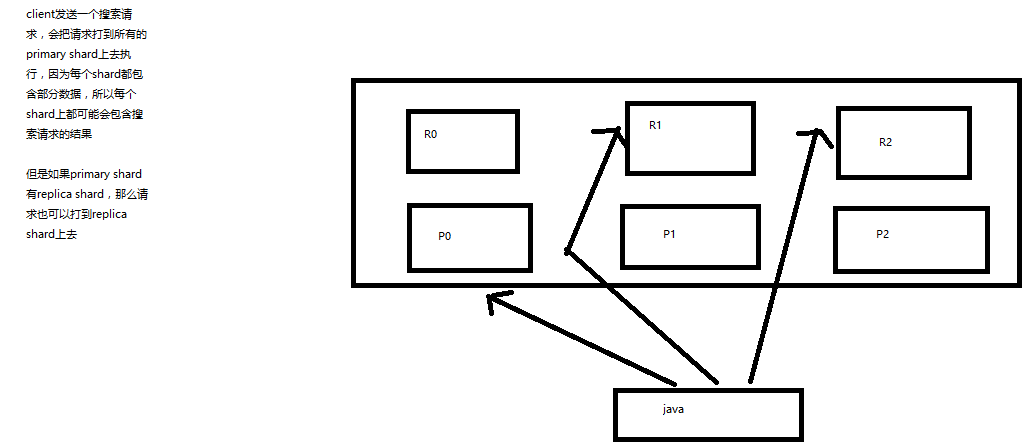
/index1/type1/\_search：搜索一个index下指定的type的数据

/index1/type1,type2/\_search：可以搜索一个index下多个type的数据

/index1,index2/type1,type2/\_search：搜索多个index下的多个type的数据

/\_all/type1,type2/\_search：\_all，可以代表搜索所有index下的指定type的数据

2、初步图解一下简单的搜索原理



## 初识搜索引擎\_分页搜索以及deep paging性能问题深度图解揭秘

1、如何使用es进行分页搜索的语法

GET /\_search?size=10

GET /\_search?size=10&from=0

GET /\_search?size=10&from=20

分页的上机实验

GET /test\_index/test\_type/\_search

"hits": {

"total": 9,

"max\_score": 1,

我们假设将这9条数据分成3页，每一页是3条数据，来实验一下这个分页搜索的效果

GET /test\_index/test\_type/\_search?from=0&size=3

第一页：id=8,6,4

GET /test\_index/test\_type/\_search?from=3&size=3

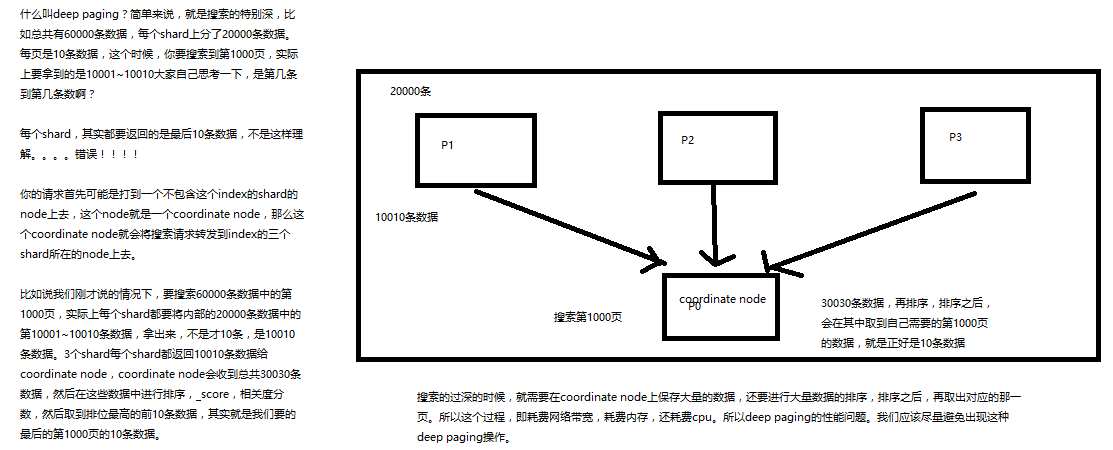
第二页：id=2,自动生成,7

GET /test\_index/test\_type/\_search?from=6&size=3

第三页：id=1,11,3

2、什么是deep paging问题？为什么会产生这个问题，它的底层原理是什么？

deep paging性能问题，以及原理深度图解揭秘，很高级的知识点



## 初识搜索引擎\_快速掌握query string search语法以及\_all metadata原理揭秘

1、query string基础语法

GET /test\_index/test\_type/\_search?q=test\_field:test //查询test\_field字段包含test的数据

GET /test\_index/test\_type/\_search?q=+test\_field:test //查询test\_field字段必须包含test的数据，和上面一样

GET /test\_index/test\_type/\_search?q=-test\_field:test//查询test\_field字段必须不包含test的数据

2、\_all metadata的原理和作用

GET /test\_index/test\_type/\_search?q=test

直接可以搜索所有的field，任意一个field包含指定的关键字就可以搜索出来。我们在进行中搜索的时候，难道是对document中的每一个field都进行一次搜索吗？不是的。

es中的\_all元数据，在建立索引的时候，我们插入一条document，它里面包含了多个field，此时，es会自动将多个field的值，全部用字符串的方式串联起来，变成一个长的字符串，作为\_all field的值，同时建立索引。

后面如果在搜索的时候，没有对某个field指定搜索，就默认搜索\_all field，其中是包含了所有field的值的。

举个例子：

{

"name": "jack",

"age": 26,

"email": "jack@sina.com",

"address": "guamgzhou"

}

"jack 26 jack@sina.com guangzhou"，作为这一条document的\_all field的值，同时进行分词后建立对应的倒排索引。

## 初识搜索引擎\_用一个例子告诉你mapping到底是什么

插入几条数据，让es自动为我们建立一个索引

PUT /website/article/1

{

"post\_date": "2017-01-01",

"title": "my first article",

"content": "this is my first article in this website",

"author\_id": 11400

}

PUT /website/article/2

{

"post\_date": "2017-01-02",

"title": "my second article",

"content": "this is my second article in this website",

"author\_id": 11400

}

PUT /website/article/3

{

"post\_date": "2017-01-03",

"title": "my third article",

"content": "this is my third article in this website",

"author\_id": 11400

}

尝试各种搜索：

GET /website/article/\_search?q=2017 3条结果

GET /website/article/\_search?q=2017-01-01 3条结果

GET /website/article/\_search?q=post\_date:2017-01-01 1条结果

GET /website/article/\_search?q=post\_date:2017 1条结果

查看es自动建立的mapping，带出什么是mapping的知识点

自动或手动为index中的type建立的一种数据结构和相关配置，简称为mapping

dynamic mapping，自动为我们建立index，创建type，以及type对应的mapping，mapping中包含了每个field对应的数据类型，以及如何分词等设置

会讲解可以手动在创建数据之前，先创建index和type，以及type对应的mapping

GET /website/\_mapping/article

{

"website": {

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "long"

},

"content": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"post\_date": {

"type": "date"

},

"title": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

搜索结果为什么不一致，因为es自动建立mapping的时候，设置了不同的field不同的data type。不同的data type的分词、搜索等行为是不一样的。所以出现了\_all field和post\_date field的搜索表现完全不一样。

## 初识搜索引擎\_精确匹配与全文搜索的对比分析

1、exact value

2017-01-01，exact value，搜索的时候，必须输入2017-01-01，才能搜索出来，如果你输入一个01，是搜索不出来的。

2、full text

（1）缩写 vs. 全程：cn vs. china

（2）格式转化：like liked likes

（3）大小写：Tom vs tom

（4）同义词：like vs love

解释：

2017-01-01，2017 01 01，搜索2017，或者01，都可以搜索出来

china，搜索cn，也可以将china搜索出来

likes，搜索like，也可以将likes搜索出来

Tom，搜索tom，也可以将Tom搜索出来

like，搜索love，同义词，也可以将like搜索出来

不是单纯的只是匹配完整的一个值，而是可以对值进行拆分词语后（分词）进行匹配，也可以通过缩写、时态、大小写、同义词等进行匹配。

## 初识搜索引擎\_倒排索引核心原理快速揭秘

doc1：I really liked my small dogs, and I think my mom also liked them.

doc2：He never liked any dogs, so I hope that my mom will not expect me to liked him.

分词，初步的倒排索引的建立

word doc1 doc2

I \* \*

really \*

liked \* \*

my \* \*

small \*

dogs \*

and \*

think \*

mom \* \*

also \*

them \*

He \*

never \*

any \*

so \*

hope \*

that \*

will \*

not \*

expect \*

me \*

to \*

him \*

演示了一下倒排索引最简单的建立的一个过程

搜索mother like little dog，不可能有任何结果

mother

like

little

dog

这个是不是我们想要的搜索结果？？？绝对不是，因为在我们看来，mother和mom有区别吗？同义词，都是妈妈的意思。like和liked有区别吗？没有，都是喜欢的意思，只不过一个是现在时，一个是过去时。little和small有区别吗？同义词，都是小小的。dog和dogs有区别吗？狗，只不过一个是单数，一个是复数。

normalization，建立倒排索引的时候，会执行一个操作，也就是说对拆分出的各个单词进行相应的处理，以提升后面搜索的时候能够搜索到相关联的文档的概率

时态的转换，单复数的转换，同义词的转换，大小写的转换

mom ―> mother

liked ―> like

small ―> little

dogs ―> dog

重新建立倒排索引，加入normalization，再次用mother liked little dog搜索，就可以搜索到了

word doc1 doc2

I \* \*

really \*

like \* \* liked --> like

my \* \*

little \* small --> little

dog \* \* dogs --> dog

and \*

think \*

mom \* \*

also \*

them \*

He \*

never \*

any \*

so \*

hope \*

that \*

will \*

not \*

expect \*

me \*

to \*

him \*

mother like little dog，分词，normalization

mother --> mom

like --> like

little --> little

dog --> dog

doc1和doc2都会搜索出来

doc1：I really liked my small dogs, and I think my mom also liked them.

doc2：He never liked any dogs, so I hope that my mom will not expect me to liked him.

## 初识搜索引擎\_分词器的内部组成到底是什么，以及内置分词器的介绍

1、分词器：切分词语，normalization（提升recall召回率）

给你一段句子，然后将这段句子拆分成一个一个的单个的单词，同时对每个单词进行normalization（时态转换，单复数转换），分词器

recall，召回率：搜索的时候，增加能够搜索到的结果的数量

词器包含部分：

character filter：在一段文本进行分词之前，先进行预处理，比如说最常见的就是，过滤html标签（<span>hello<span> --> hello），（I&you --> I and you）

tokenizer：分词，hello you and me --> hello, you, and, me

token filter：lowercase，stop word，synonymom，dogs --> dog，liked --> like，Tom --> tom，a/the/an --> 干掉，mother --> mom，small --> little

一个分词器，很重要，将一段文本进行各种处理，最后处理好的结果才会拿去建立倒排索引

2、内置分词器的介绍

Set the shape to semi-transparent by calling set\_trans(5)

standard analyzer：set, the, shape, to, semi, transparent, by, calling, set\_trans, 5（默认的是standard）

simple analyzer：set, the, shape, to, semi, transparent, by, calling, set, trans

whitespace analyzer：Set, the, shape, to, semi-transparent, by, calling, set\_trans(5)

language analyzer（特定的语言的分词器，比如说，english，英语分词器）：set, shape, semi, transpar, call, set\_tran, 5

## 初识搜索引擎\_query string的分词以及mapping引入案例遗留问题的大揭秘

1、query string分词

query string必须以和index建立时相同的analyzer进行分词

query string对exact value和full text的区别对待

date：exact value

\_all：full text

比如我们有一个document，其中有一个field，包含的value是：hello you and me，建立倒排索引,我们要搜索这个document对应的index，搜索文本是hell me，这个搜索文本就是query string，默认情况下，es会使用它对应的field建立倒排索引时相同的分词器去进行分词，分词和normalization，只有这样，才能实现正确的搜索

我们建立倒排索引的时候，将dogs --> dog，结果你搜索的时候，还是一个dogs，那不就搜索不到了吗？所以搜索的时候，那个dogs也必须变成dog才行。才能搜索到。

不同类型的field，可能有的就是full text，有的就是exact value

post\_date，date：exact value

\_all：full text，分词，normalization

2、mapping引入案例遗留问题大揭秘

GET /\_search?q=2017

搜索的是\_all field，document所有的field都会拼接成一个大串，进行分词

2017-01-02 my second article this is my second article in this website 11400

doc1 doc2 doc3

2017 \* \* \*

01 \*

02 \*

03 \*

\_all，2017，自然会搜索到3个docuemnt

GET /\_search?q=2017-01-01

\_all，2017-01-01，query string会用跟建立倒排索引一样的分词器去进行分词

2017

01

01

GET /\_search?q=post\_date:2017-01-01

date，会作为exact value去建立索引

doc1 doc2 doc3

2017-01-01 \*

2017-01-02 \*

2017-01-03 \*

post\_date:2017-01-01，2017-01-01，doc1一条document

GET /\_search?q=post\_date:2017，这个在这里不讲解，因为是es 5.2以后做的一个优化

3、测试分词器

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

## 初识搜索引擎\_什么是mapping再次回炉透彻理解

（1）往es里面直接插入数据，es会自动建立索引，同时建立type以及对应的mapping

（2）mapping中就自动定义了每个field的数据类型

（3）不同的数据类型（比如说text和date），可能有的是exact value，有的是full text

（4）exact value，在建立倒排索引的时候，分词的时候，是将整个值一起作为一个关键词建立到倒排索引中的；full text，会经历各种各样的处理，分词，normaliztion（时态转换，同义词转换，大小写转换），才会建立到倒排索引中

（5）exact value和full text类型的field决定了搜索时，对exact value field或者是full text field进行搜索的行为，会跟建立倒排索引的行为保持一致；比如说exact value搜索的时候，就是直接按照整个值进行匹配，full text query string，也会进行分词和normalization再去倒排索引中去搜索

（6）可以用es的dynamic mapping，让其自动建立mapping，包括自动设置数据类型；也可以提前手动创建index和type的mapping，自己对各个field进行设置，包括数据类型，包括索引行为，包括分词器，等等

mapping，就是index的type的元数据，每个type都有一个自己的mapping，决定了数据类型，建立倒排索引的行为，还有进行搜索的行为

## 初识搜索引擎\_mapping的核心数据类型以及dynamic mapping

1、核心的数据类型

string

byte，short，integer，long

float，double

boolean

date

2、dynamic mapping

true or false --> boolean

123 --> long

123.45 --> double

2017-01-01 --> date

"hello world" --> string/text

3、查看mapping

GET /index/\_mapping/type

## 初识搜索引擎\_手动建立和修改mapping以及定制string类型数据是否分词

1、如何建立索引

Analyzed、not\_analyzed（extra value）、no（不被索引和搜索）

2、修改mapping

只能创建index时手动建立mapping，或者新增field mapping，但是不能update field mapping

新建：

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "long"

},

"title": {

"type": "text",

"analyzer": "english"

},

"content": {

"type": "text"

},

"post\_date": {

"type": "date"

},

"publisher\_id": {

"type": "text",

"index": "not\_analyzed"

}

}

}

}

}

无法修改字段：

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "text"

}

}

}

}

}

报错如下：

{

"error": {

"root\_cause": [

{

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

"reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",

"index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",

"index": "website"

}

],

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

"reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",

"index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",

"index": "website"

},

"status": 400

}

新加字段：

PUT /website/\_mapping/article

{

"properties" : {

"new\_field" : {

"type" : "string",

"index": "not\_analyzed"

}

}

}

3、测试mapping

GET /website/\_analyze

{

"field": "content",

"text": "my-dogs"

}

GET website/\_analyze

{

"field": "new\_field",

"text": "my dogs"

}

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[4onsTYV][127.0.0.1:9300][indices:admin/analyze[s]]"

}

],

"type": "illegal\_argument\_exception",

"reason": "Can't process field [new\_field], Analysis requests are only supported on tokenized fields"

},

"status": 400

}

## 初识搜索引擎\_mapping复杂数据类型以及object类型数据底层结构大揭秘

1、multivalue field

{ "tags": [ "tag1", "tag2" ]}

建立索引时与string是一样的，数据类型不能混合

2、empty field

null，[]，[null]

3、object field

PUT /company/employee/1

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

address：object类型

{

"company": {

"mappings": {

"employee": {

"properties": {

"address": {

"properties": {

"city": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"country": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"province": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

},

"age": {

"type": "long"

},

"join\_date": {

"type": "date"

},

"name": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

上面的会转为下面这样：

{

"name": [jack],

"age": [27],

"join\_date": [2017-01-01],

"address.country": [china],

"address.province": [guangdong],

"address.city": [guangzhou]

}

{

"authors": [

{ "age": 26, "name": "Jack White"},

{ "age": 55, "name": "Tom Jones"},

{ "age": 39, "name": "Kitty Smith"}

]

}

上面的会转为下面这样：

{

"authors.age": [26, 55, 39],

"authors.name": [jack, white, tom, jones, kitty, smith]

}