# 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

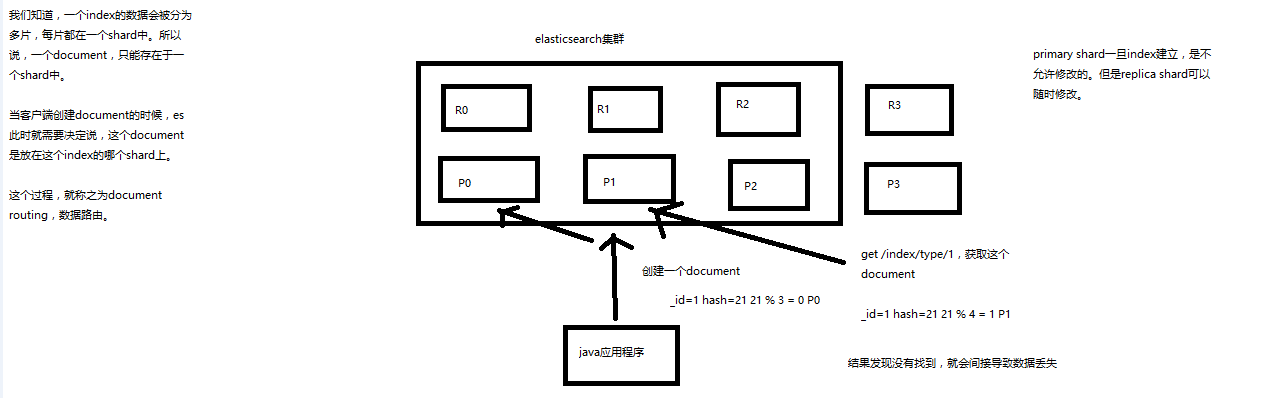
无论hash值是几，无论是什么数字，对number\_of\_primary\_shards求余数，结果一定是在0~number\_of\_primary\_shards-1之间这个范围内的。

### \_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



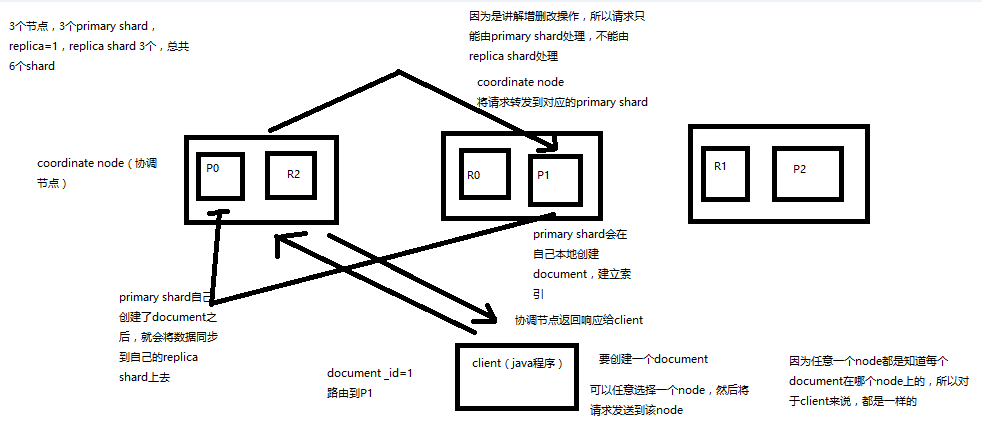
### 分布式文档系统\_document增删改内部原理图解揭秘

（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

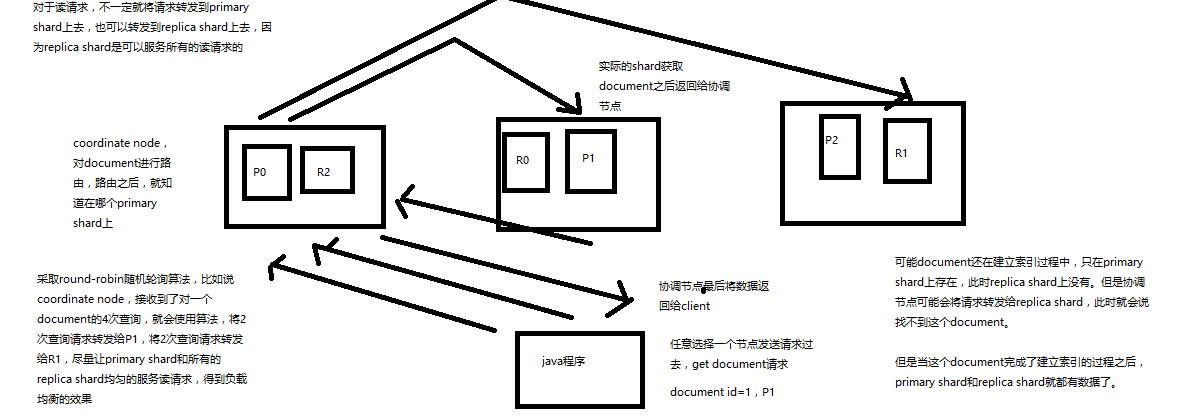
1、客户端发送请求到任意一个node，成为coordinate node

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

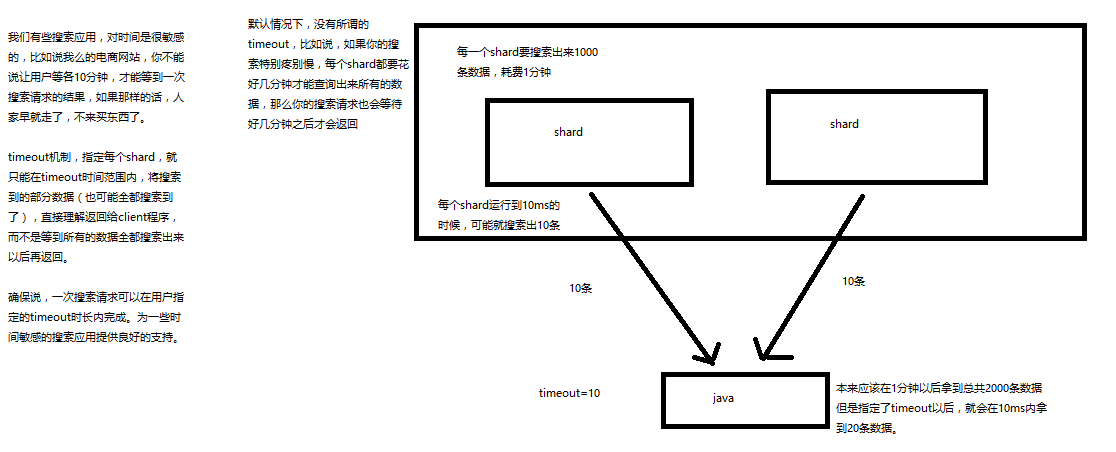
shards：shards fail的条件（primary和replica全部挂掉），不影响其他shard。默认情况下来说，一个搜索请求，会打到一个index的所有primary shard上去，当然了，每个primary shard都可能会有一个或多个replic shard，所以请求也可以到primary shard的其中一个replica shard上去。

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

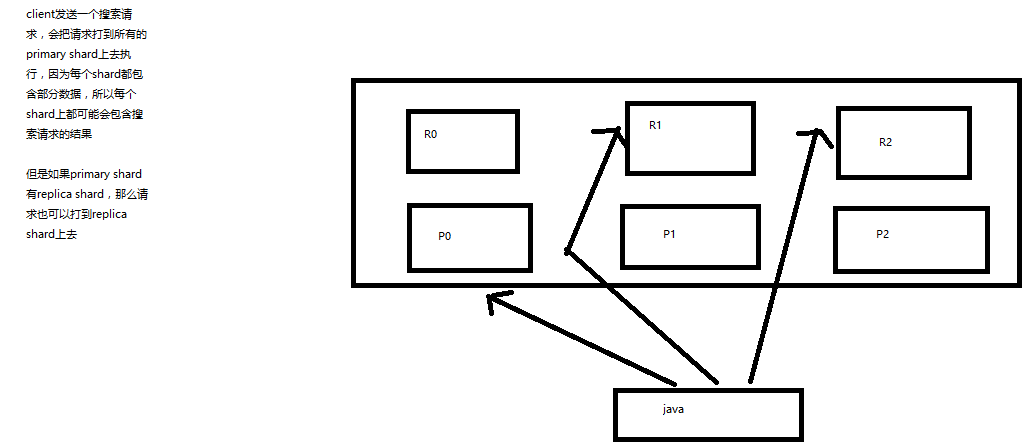
/index1/type1/\_search：搜索一个index下指定的type的数据

/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

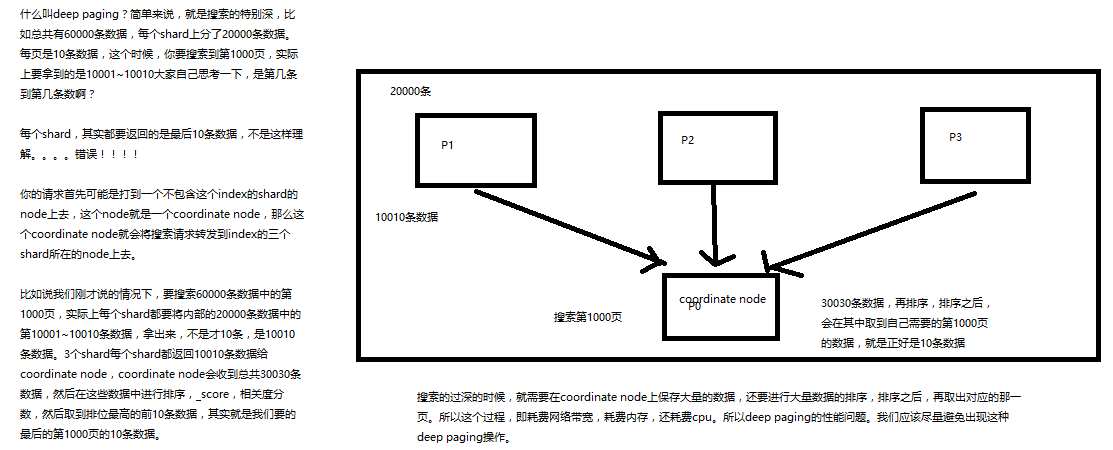
第二页：id=2,自动生成,7

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]

}

## 初识搜索引擎\_search api的基础语法介绍

1、search api的基本语法

GET /search

{}

GET /index1,index2/type1,type2/search

{}

GET /\_search

{

"from": 0,

"size": 10

}

2、http协议中get是否可以带上request body

HTTP协议，一般不允许get请求带上request body，但是因为get更加适合描述查询数据的操作，因此还是这么用了

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

POST /\_search

{

"from":0,

"size":10

}

碰巧，很多浏览器，或者是服务器，也都支持GET+request body模式

如果遇到不支持的场景，也可以用POST /\_search

## 初识搜索引擎\_快速上机动手实战Query DSL搜索语法

1、一个例子让你明白什么是Query DSL

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、Query DSL的基本语法

{

QUERY\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

{

QUERY\_NAME: {

FIELD\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

}

示例：

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

{

"query": {

"match": {

"test\_field": "test"

}

}

}

3、如何组合多个搜索条件

搜索需求：title必须包含elasticsearch，content可以包含elasticsearch也可以不包含，author\_id必须不为111

{

"took": 1,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "website",

"\_type": "article",

"\_id": "2",

"\_score": 1,

"\_source": {

"title": "my hadoop article",

"content": "hadoop is very bad",

"author\_id": 111

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "my elasticsearch article",

"content": "es is very bad",

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "my elasticsearch article",

"content": "es is very goods",

"author\_id": 111

}

}

]

}

}

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

GET /test\_index/\_search

{

"query": {

"bool": {

"must": { "match": { "name": "tom" }},

"should": [

{ "match": { "hired": true }},

{ "bool": {

"must": { "match": { "personality": "good" }},

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

}}

],

"minimum\_should\_match": 1

}

}

}

## 初识搜索引擎\_filter与query深入对比解密：相关度，性能

1、filter与query示例

PUT /company/employee/2

{

"address": {

"country": "china",

"province": "jiangsu",

"city": "nanjing"

},

"name": "tom",

"age": 30,

"join\_date": "2016-01-01"

}

PUT /company/employee/3

{

"address": {

"country": "china",

"province": "shanxi",

"city": "xian"

},

"name": "marry",

"age": 35,

"join\_date": "2015-01-01"

}

搜索请求：年龄必须大于等于30，同时join\_date必须是2016-01-01

GET /company/employee/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"join\_date": "2016-01-01"

}

}

],

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

}

}

2、filter与query对比大解密

filter，仅仅只是按照搜索条件过滤出需要的数据而已，不计算任何相关度分数，对相关度没有任何影响

query，会去计算每个document相对于搜索条件的相关度，并按照相关度进行排序

一般来说，如果你是在进行搜索，需要将最匹配搜索条件的数据先返回，那么用query；如果你只是要根据一些条件筛选出一部分数据，不关注其排序，那么用filter

除非是你的这些搜索条件，你希望越符合这些搜索条件的document越排在前面返回，那么这些搜索条件要放在query中；如果你不希望一些搜索条件来影响你的document排序，那么就放在filter中即可

3、filter与query性能

filter，不需要计算相关度分数，不需要按照相关度分数进行排序，同时还有内置的自动cache最常使用filter的数据

query，相反，要计算相关度分数，按照分数进行排序，而且无法cache结果

## 初识搜索引擎\_上机动手实战常用的各种query搜索语法

1、match all

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、match

GET /\_search

{

"query": { "match": { "title": "my elasticsearch article" }}

}

3、multi match 多个字段其中一个字段匹配某个值都满足

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

{

"query": {

"multi\_match": {

"query": "test",

"fields": ["test\_field", "test\_field1"]

}

}

}

4、range query

GET /company/employee/\_search

{

"query": {

"range": {

"age": {

"gte": 30

}

}

}

}

5、term query把这个字段当做整个串（不分词）进行查询，也可以不用term，建mapping的时候设置为不分词就行。

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

{

"query": {

"term": {

"test\_field": "test hello"

}

}

}

6、terms query 对某个字段指定多个搜索词

GET /\_search

{

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

}

## 初识搜索引擎\_上机动手实战多搜索条件组合查询

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

{

"bool": {

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

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

"should": [

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

],

"filter": {

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

}

}

}

bool

must，must\_not，should，filter

must: 文档必须完全匹配条件

should: should下面会带一个以上的条件，至少满足一个条件，这个文档就符合should

must\_not: 文档必须不匹配条件

每个子查询都会计算一个document针对它的相关度分数，然后bool综合所有分数，合并为一个分数，当然filter是不会计算分数的

{

"bool": {

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

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

"should": [

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

],

"filter": {

"bool": {

"must": [

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

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

],

"must\_not": [

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

]

}

}

}

}

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

}

}

## 初识搜索引擎\_上机动手实战如何定位不合法的搜索以及其原因

一般用在那种特别复杂庞大的搜索下，比如你一下子写了上百行的搜索，这个时候可以先用validate api去验证一下，搜索是否合法。

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"math": {

"test\_field": "test"

}

}

}

{

"valid": false,

"error": "org.elasticsearch.common.ParsingException: no [query] registered for [math]"

}

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"match": {

"test\_field": "test"

}

}

}

{

"valid": true,

"\_shards": {

"total": 1,

"successful": 1,

"failed": 0

},

"explanations": [

{

"index": "test\_index",

"valid": true,

"explanation": "+test\_field:test #(#\_type:test\_type)"

}

]

}

## 初识搜素引擎\_上机动手实战如何定制搜索结果的排序规则

1、默认情况下，是按照\_score降序排序的

然而，某些情况下，可能没有有用的\_score，比如说filter

GET /\_search

{

"query" : {

"bool" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

当然，也可以是constant\_score

GET /\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

2、定制排序规则

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

},

"sort": [

{

"join\_date": {

"order": "asc"

}

}

]

}

## 初识搜索引擎\_解密如何将一个field索引两次来解决字符串排序问题

如果对一个string field进行排序，结果往往不准确，因为分词后是多个单词，再排序就不是我们想要的结果了

通常解决方案是，将一个string field建立两次索引，一个分词，用来进行搜索；一个不分词，用来进行排序

PUT /website

{

"mappings": {

"article": {

"properties": {

"title": {

"type": "text",

"fields": {

"raw": {

"type": "string",

"index": "not\_analyzed"

}

},

"fielddata": true

},

"content": {

"type": "text"

},

"post\_date": {

"type": "date"

},

"author\_id": {

"type": "long"

}

}

}

}

}

PUT /website/article/1

{

"title": "first article",

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

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

"author\_id": 110

}

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "website",

"\_type": "article",

"\_id": "2",

"\_score": 1,

"\_source": {

"title": "first article",

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

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

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "second article",

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

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

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "third article",

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

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

"author\_id": 110

}

}

]

}

}

GET /website/article/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"title.raw": {

"order": "desc"

}

}

]

}

## 初识搜索引擎\_相关度评分TF&IDF算法独家解密

1、算法介绍

relevance score算法，简单来说，就是计算出，一个索引中的文本，与搜索文本，他们之间的关联匹配程度

Elasticsearch使用的是 term frequency/inverse document frequency算法，简称为TF/IDF算法

**Term frequency：搜索文本中的各个词条在field文本中出现了多少次，出现次数越多，就越相关**

比如，搜索请求：hello world

doc1：hello you, and world is very good

doc2：hello, how are you

**Inverse document frequency：搜索文本中的各个词条在整个索引的所有文档中出现了多少次，出现的次数越多，就越不相关**

比如搜索请求：hello world

doc1：hello, today is very good

doc2：hi world, how are you

比如说，在index中有1万条document，hello这个单词在所有的document中，一共出现了1000次；world这个单词在所有的document中，一共出现了100次

doc2更相关

**Field-length norm：field长度，field越长，相关度越弱**

搜索请求：hello world

doc1：{ "title": "hello article", "content": "babaaba 1万个单词" }

doc2：{ "title": "my article", "content": "blablabala 1万个单词，hi world" }

hello world在整个index中出现的次数是一样多的

doc1更相关，title field更短

2、\_score是如何被计算出来的

GET /test\_index/test\_type/\_search?explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}

3、分析一个document是如何被匹配上的

GET /test\_index/test\_type/6/\_explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}

## 初识搜索引擎\_内核级知识点之doc value初步探秘

搜索的时候，要依靠倒排索引；排序的时候，需要依靠正排索引，看到每个document的每个field，然后进行排序，所谓的正排索引，其实就是doc values

在建立索引的时候，一方面会建立倒排索引，以供搜索用；一方面会建立正排索引，也就是doc values，以供排序，聚合，过滤等操作使用

doc values是被保存在磁盘上的，此时如果内存足够，os会自动将其缓存在内存中，性能还是会很高；如果内存不足够，os会将其写入磁盘上

doc1: hello world you and me

doc2: hi, world, how are you

word doc1 doc2

hello \*

world \* \*

you \* \*

and \*

me \*

hi \*

how \*

are \*

hello you --> hello, you

hello --> doc1

you --> doc1,doc2

doc1: hello world you and me

doc2: hi, world, how are you

sort by age

doc1: { "name": "jack", "age": 27 }

doc2: { "name": "tom", "age": 30 }

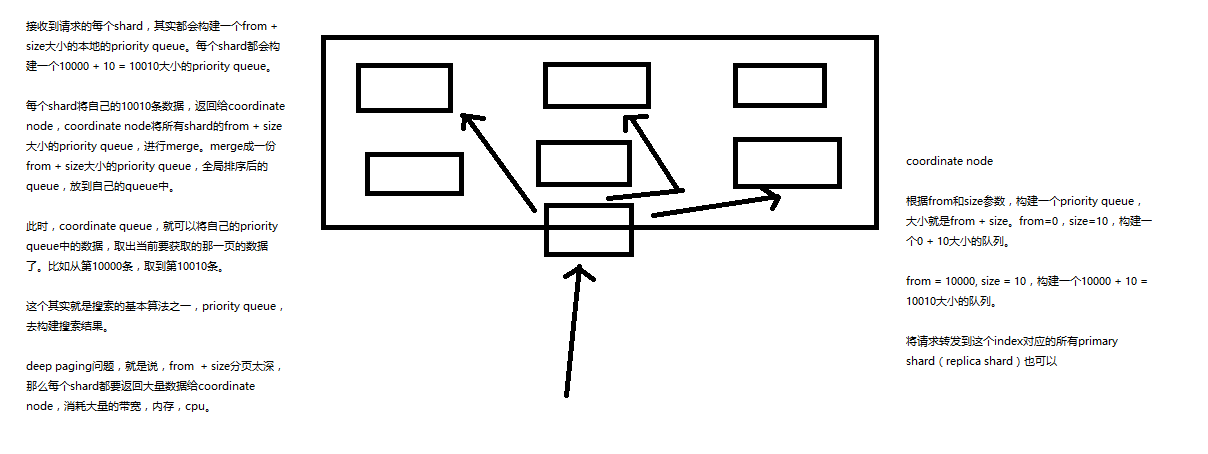
document name age

doc1 jack 27

doc2 tom 30

## 初识搜索引擎\_分布式搜索引擎内核解密之query phase

课程大纲



1、query phase

（1）搜索请求发送到某一个coordinate node，构建一个priority queue，长度以paging操作from和size为准，默认为10

（2）coordinate node将请求转发到所有shard，每个shard本地搜索，并构建一个本地的priority queue

（3）各个shard将自己的priority queue返回给coordinate node，并构建一个全局的priority queue

2、replica shard如何提升搜索吞吐量

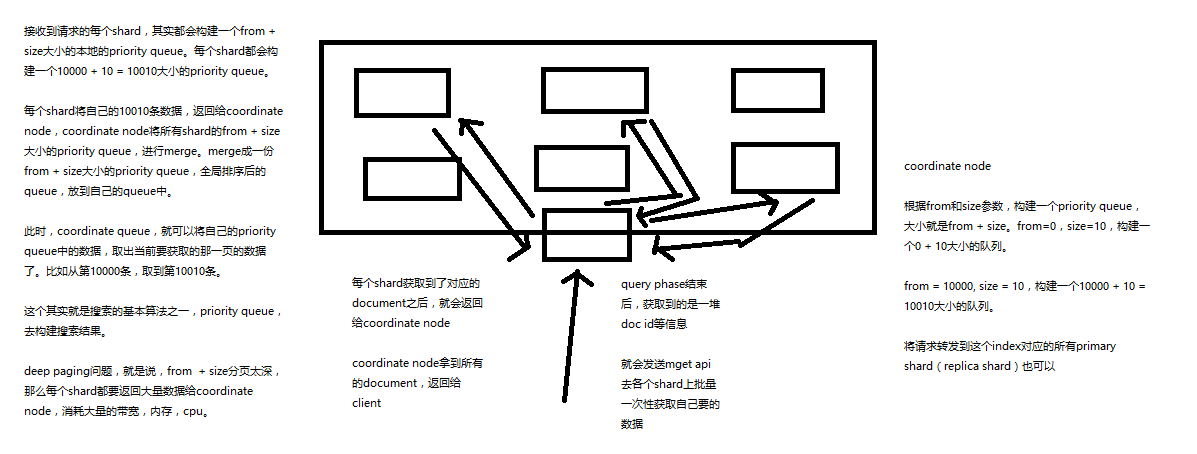
一次请求要打到所有shard的一个replica/primary上去，如果每个shard都有多个replica，那么同时并发过来的搜索请求可以同时打到其他的replica上去

## fetch phase工作流程

（1）coordinate node构建完priority queue之后，就发送mget请求去所有shard上获取对应的document

（2）各个shard将document返回给coordinate node

（3）coordinate node将合并后的document结果返回给client客户端



一般搜索，如果不加from和size，就默认搜索前10条，按照\_score排序

## 初识搜索引擎\_搜索相关参数梳理以及bouncing results问题解决方案

1、preference

决定了哪些shard会被用来执行搜索操作

\_primary, \_primary\_first, \_local, \_only\_node:xyz, \_prefer\_node:xyz, \_shards:2,3

bouncing results问题，两个document排序，field值相同；不同的shard上，可能排序不同；每次请求轮询打到不同的replica shard上；每次页面上看到的搜索结果的排序都不一样。这就是bouncing result，也就是跳跃的结果。

搜索的时候，是轮询将搜索请求发送到每一个replica shard（primary shard），但是在不同的shard上，可能document的排序不同,解决方案就是将preference设置为一个字符串，比如说user\_id，让每个user每次搜索的时候，都使用同一个replica shard去执行，就不会看到bouncing results了

2、timeout，已经讲解过原理了，主要就是限定在一定时间内，将部分获取到的数据直接返回，避免查询耗时过长

3、routing，document文档路由，\_id路由，routing=user\_id，这样的话可以让同一个user对应的数据到一个shard上去

4、search\_type

default：query\_then\_fetch

dfs\_query\_then\_fetch，可以提升revelance sort精准度

## 初识搜索引擎\_上机动手实战基于scoll技术滚动搜索大量数据

如果一次性要查出来比如10万条数据，那么性能会很差，此时一般会采取用scoll滚动查询，一批一批的查，直到所有数据都查询完处理完,使用scoll滚动搜索，可以先搜索一批数据，然后下次再搜索一批数据，以此类推，直到搜索出全部的数据来,scoll搜索会在第一次搜索的时候，保存一个当时的视图快照，之后只会基于该旧的视图快照提供数据搜索，如果这个期间数据变更，是不会让用户看到的。

采用基于\_doc进行排序的方式，性能较高

每次发送scroll请求，我们还需要指定一个scroll参数，指定一个时间窗口，每次搜索请求只要在这个时间窗口内能完成就可以了

GET /test\_index/test\_type/\_search?scroll=1m

{

"query": {

"match\_all": {}

},

"sort": [ "\_doc" ],

"size": 3

}

{

"\_scroll\_id": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACxeFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAALF8WNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACxhFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYhY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 10,

"max\_score": null,

"hits": [

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_score": null,

"\_source": {

"test\_field": "test client 2"

},

"sort": [

0

]

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_score": null,

"\_source": {

"test\_field": "tes test"

},

"sort": [

0

]

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "AVp4RN0bhjxldOOnBxaE",

"\_score": null,

"\_source": {

"test\_content": "my test"

},

"sort": [

0

]

}

]

}

}

获得的结果会有一个scoll\_id，下一次再发送scoll请求的时候，必须带上这个scoll\_id

GET /\_search/scroll

{

"scroll": "1m",

"scroll\_id" : "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACxeFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAALF8WNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACxhFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYhY0b25zVFlWWlRqR3ZJajlfc3BXejJ3"

}

scoll，看起来挺像分页的，但是其实使用场景不一样。分页主要是用来一页一页搜索，给用户看的；scoll主要是用来一批一批检索数据，让系统进行处理的

## 索引管理\_快速上机动手实战创建、修改以及删除索引

1、为什么我们要手动创建索引？

2、创建索引

创建索引的语法

PUT /my\_index

{

"settings": { ... any settings ... },

"mappings": {

"type\_one": { ... any mappings ... },

"type\_two": { ... any mappings ... },

...

}

}

创建索引的示例

PUT /my\_index

{

"settings": {

"number\_of\_shards": 1,

"number\_of\_replicas": 0

},

"mappings": {

"my\_type": {

"properties": {

"my\_field": {

"type": "text"

}

}

}

}

}

3、修改索引

PUT my\_index/\_settings

{

"number\_of\_replicas": 1

}

4、删除索引

DELETE /my\_index

DELETE /index\_one,index\_two

DELETE /index\_\*

DELETE /\_all

elasticsearch.yml

action.destructive\_requires\_name: true//指定后不能使用DELETE /\_all干掉所有索引

## 索引管理\_快速上机动手实战修改分词器以及定制自己的分词器

1、默认的分词器

standard

standard tokenizer：以单词边界进行切分

standard token filter：什么都不做

lowercase token filter：将所有字母转换为小写

stop token filer（默认被禁用）：移除停用词，比如a the it等等

2、修改分词器的设置

启用english停用词token filter

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"es\_std": {

"type": "standard",

"stopwords": "\_english\_"

}

}

}

}

}

测试：

GET /my\_index/\_analyze

{

"analyzer": "standard",

"text": "a dog is in the house"

}

GET /my\_index/\_analyze

{

"analyzer": "es\_std",

"text":"a dog is in the house"

}

3、定制化自己的分词器

PUT /my\_index

{

"settings": {

"analysis": {

"char\_filter": {

"&\_to\_and": {

"type": "mapping",

"mappings": ["&=> and"]

}

},

"filter": {

"my\_stopwords": {

"type": "stop",

"stopwords": ["the", "a"]

}

},

"analyzer": {

"my\_analyzer": {

"type": "custom",

"char\_filter": ["html\_strip", "&\_to\_and"],

"tokenizer": "standard",

"filter": ["lowercase", "my\_stopwords"]

}

}

}

}

}

测试

GET /my\_index/\_analyze

{

"text": "tom&jerry are a friend in the house, <a>, HAHA!!",

"analyzer": "my\_analyzer"

}

在type里使用自定义分词器

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

{

"properties": {

"content": {

"type": "text",

"analyzer": "my\_analyzer"

}

}

}

## 索引管理\_内核级知识点：深入探秘type底层数据结构

type，是一个index中用来区分类似的数据的，类似的数据，但是可能有不同的fields，而且有不同的属性来控制索引建立、分词器

field的value，在底层的lucene中建立索引的时候，全部是opaque bytes类型，不区分类型的

lucene是没有type的概念的，在document中，实际上将type作为一个document的field来存储，即\_type，es通过\_type来进行type的过滤和筛选

一个index中的多个type，实际上是放在一起存储的，因此一个index下，不能有多个type重名，而类型或者其他设置不同的，因为那样是无法处理的

{

"ecommerce": {

"mappings": {

"elactronic\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"service\_period": {

"type": "string"

}

}

},

"fresh\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"eat\_period": {

"type": "string"

}

}

}

}

}

}

{

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year"

}

{

"name": "aozhou dalongxia",

"price": 199.0,

"eat\_period": "one week"

}

在底层的存储是这样子的。。。。

{

"ecommerce": {

"mappings": {

"\_type": {

"type": "string",

"index": "not\_analyzed"

},

"name": {

"type": "string"

}

"price": {

"type": "double"

}

"service\_period": {

"type": "string"

}

"eat\_period": {

"type": "string"

}

}

}

}

{

"\_type": "elactronic\_goods",

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year",

"eat\_period": ""

}

{

"\_type": "fresh\_goods",

"name": "aozhou dalongxia",

"price": 199.0,

"service\_period": "",

"eat\_period": "one week"

}

最佳实践，将类似结构的type放在一个index下，这些type应该有多个field是相同的

假如说，你将两个type的field完全不同，放在一个index下，那么就每条数据都至少有一半的field在底层的lucene中是空值，会有严重的性能问题

## 索引管理\_mapping root object深入剖析

1、root object

就是某个type对应的mapping json，包括了properties，metadata（\_id，\_source，\_type），settings（analyzer），其他settings（比如include\_in\_all）

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {}

}

}

}

2、properties

type，index，analyzer

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

{

"properties": {

"title": {

"type": "text"

}

}

}

3、\_source

好处

（1）查询的时候，直接可以拿到完整的document，不需要先拿document id，再发送一次请求拿document

（2）partial update基于\_source实现

（3）reindex时，直接基于\_source实现，不需要从数据库（或者其他外部存储）查询数据再修改

（4）可以基于\_source定制返回field

（5）debug query更容易，因为可以直接看到\_source

如果不需要上述好处，可以禁用\_source

PUT /my\_index/\_mapping/my\_type2

{

"\_source": {"enabled": false}

}

4、\_all

将所有field打包在一起，作为一个\_all field，建立索引。没指定任何field进行搜索时，就是使用\_all field在搜索。

PUT /my\_index/\_mapping/my\_type3

{

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

}

也可以在field级别设置include\_in\_all field，设置是否要将field的值包含在\_all field中

PUT /my\_index/\_mapping/my\_type4

{

"properties": {

"my\_field": {

"type": "text",

"include\_in\_all": false

}

}

}

5、标识性metadata

\_index，\_type，\_id

## 索引管理\_定制化自己的dynamic mapping策略

1、定制dynamic策略

true：遇到陌生字段，就进行dynamic mapping

false：遇到陌生字段，就忽略

strict：遇到陌生字段，就报错

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic": "strict",

"properties": {

"title": {

"type": "text"

},

"address": {

"type": "object",

"dynamic": "true"

}

}

}

}

}

PUT /my\_index/my\_type/1

{

"title": "my article",

"content": "this is my article",

"address": {

"province": "guangdong",

"city": "guangzhou"

}

}

{

"error": {

"root\_cause": [

{

"type": "strict\_dynamic\_mapping\_exception",

"reason": "mapping set to strict, dynamic introduction of [content] within [my\_type] is not allowed"

}

],

"type": "strict\_dynamic\_mapping\_exception",

"reason": "mapping set to strict, dynamic introduction of [content] within [my\_type] is not allowed"

},

"status": 400

}

PUT /my\_index/my\_type/1

{

"title": "my article",

"address": {

"province": "guangdong",

"city": "guangzhou"

}

}

GET /my\_index/\_mapping/my\_type

{

"my\_index": {

"mappings": {

"my\_type": {

"dynamic": "strict",

"properties": {

"address": {

"dynamic": "true",

"properties": {

"city": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"province": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

},

"title": {

"type": "text"

}

}

}

}

}

}

2、定制dynamic mapping策略

（1）date\_detection

默认会按照一定格式识别date，比如yyyy-MM-dd。但是如果某个field先过来一个2017-01-01的值，就会被自动dynamic mapping成date，后面如果再来一个"hello world"之类的值，就会报错。可以手动关闭某个type的date\_detection，如果有需要，自己手动指定某个field为date类型。

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

{

"date\_detection": false

}

（2）定制自己的dynamic mapping template（type level）

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{ "en": {

"match": "\*\_en",

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

"mapping": {

"type": "string",

"analyzer": "english"

}

}}

]

}}}

PUT /my\_index/my\_type/1

{

"title": "this is my first article"

}

PUT /my\_index/my\_type/2

{

"title\_en": "this is my first article"

}

title没有匹配到任何的dynamic模板，默认就是standard分词器，不会过滤停用词，is会进入倒排索引，用is来搜索是可以搜索到的

title\_en匹配到了dynamic模板，就是english分词器，会过滤停用词，is这种停用词就会被过滤掉，用is来搜索就搜索不到了

（3）定制自己的default mapping template（index level）

PUT /my\_index

{

"mappings": {

"\_default\_": {

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

},

"blog": {

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

}

}

}

## 索引管理\_复杂上机实验：基于scoll+bulk+索引别名实现零停机重建索引

1、重建索引

一个field的设置是不能被修改的，如果要修改一个Field，那么应该重新按照新的mapping，建立一个index，然后将数据批量查询出来，重新用bulk api写入index中

批量查询的时候，建议采用scroll api，并且采用多线程并发的方式来reindex数据，每次scoll就查询指定日期的一段数据，交给一个线程即可

（1）一开始，依靠dynamic mapping，插入数据，但是不小心有些数据是2017-01-01这种日期格式的，所以title这种field被自动映射为了date类型，实际上它应该是string类型的

PUT /my\_index/my\_type/3

{

"title": "2017-01-03"

}

{

"my\_index": {

"mappings": {

"my\_type": {

"properties": {

"title": {

"type": "date"

}

}

}

}

}

}

（2）当后期向索引中加入string类型的title值的时候，就会报错

PUT /my\_index/my\_type/4

{

"title": "my first article"

}

{

"error": {

"root\_cause": [

{

"type": "mapper\_parsing\_exception",

"reason": "failed to parse [title]"

}

],

"type": "mapper\_parsing\_exception",

"reason": "failed to parse [title]",

"caused\_by": {

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

"reason": "Invalid format: \"my first article\""

}

},

"status": 400

}

（3）如果此时想修改title的类型，是不可能的

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

{

"properties": {

"title": {

"type": "text"

}

}

}

{

"error": {

"root\_cause": [

{

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

"reason": "mapper [title] of different type, current\_type [date], merged\_type [text]"

}

],

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

"reason": "mapper [title] of different type, current\_type [date], merged\_type [text]"

},

"status": 400

}

（4）此时，唯一的办法，就是进行reindex，也就是说，重新建立一个索引，将旧索引的数据查询出来，再导入新索引

（5）如果说旧索引的名字，是old\_index，新索引的名字是new\_index，终端java应用，已经在使用old\_index在操作了，难道还要去停止java应用，修改使用的index为new\_index，才重新启动java应用吗？这个过程中，就会导致java应用停机，可用性降低

（6）所以说，给java应用一个别名，这个别名是指向旧索引的，java应用先用着，java应用先用goods\_index alias来操作，此时实际指向的是旧的my\_index

PUT /my\_index/\_alias/goods\_index

（7）新建一个index，调整其title的类型为string

PUT /my\_index\_new

{

"mappings": {

"my\_type": {

"properties": {

"title": {

"type": "text"

}

}

}

}

}

（8）使用scroll api将数据批量查询出来

GET /my\_index/\_search?scroll=1m

{

"query": {

"match\_all": {}

},

"sort": ["\_doc"],

"size": 1

}

{

"\_scroll\_id": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAADpAFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6QRY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAAOkIWNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAADpDFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6RBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",

"took": 1,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": null,

"hits": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "2",

"\_score": null,

"\_source": {

"title": "2017-01-02"

},

"sort": [

0

]

}

]

}

}

（9）采用bulk api将scoll查出来的一批数据，批量写入新索引

POST /\_bulk

{ "index": { "\_index": "my\_index\_new", "\_type": "my\_type", "\_id": "2" }}

{ "title": "2017-01-02" }

（10）反复循环8~9，查询一批又一批的数据出来，采取bulk api将每一批数据批量写入新索引

（11）将goods\_index alias切换到my\_index\_new上去，java应用会直接通过index别名使用新的索引中的数据，java应用程序不需要停机，高可用

POST /\_aliases

{

"actions": [

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

{ "add": { "index": "my\_index\_new", "alias": "goods\_index" }}

]

}

（12）直接通过goods\_index别名来查询，是否ok

GET /goods\_index/my\_type/\_search

2、基于alias对client透明切换index

PUT /my\_index\_v1/\_alias/my\_index

client对my\_index进行操作

reindex操作，完成之后，切换v1到v2

POST /\_aliases

{

"actions": [

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

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

]

}

## 内核原理探秘\_倒排索引组成结构以及其索引可变原因揭秘

倒排索引，是适合用于进行搜索的

倒排索引的结构

（1）包含这个关键词的document list

（2）包含这个关键词的所有document的数量：IDF（inverse document frequency）

（3）这个关键词在每个document中出现的次数：TF（term frequency）

（4）这个关键词在这个document中的次序

（5）每个document的长度：length norm

（6）包含这个关键词的所有document的平均长度

word doc1 doc2

dog \* \*

hello \*

you \*

倒排索引不可变的好处

（1）不需要锁，提升并发能力，避免锁的问题

（2）数据不变，一直保存在os cache中，只要cache内存足够

（3）filter cache一直驻留在内存，因为数据不变

（4）可以压缩，节省cpu和io开销

倒排索引不可变的坏处：每次都要重新构建整个索引

## 内核原理探秘\_深度图解剖析document写入原理（buffer，segment，commit）

（1）数据写入buffer

（2）commit point

（3）buffer中的数据写入新的index segment

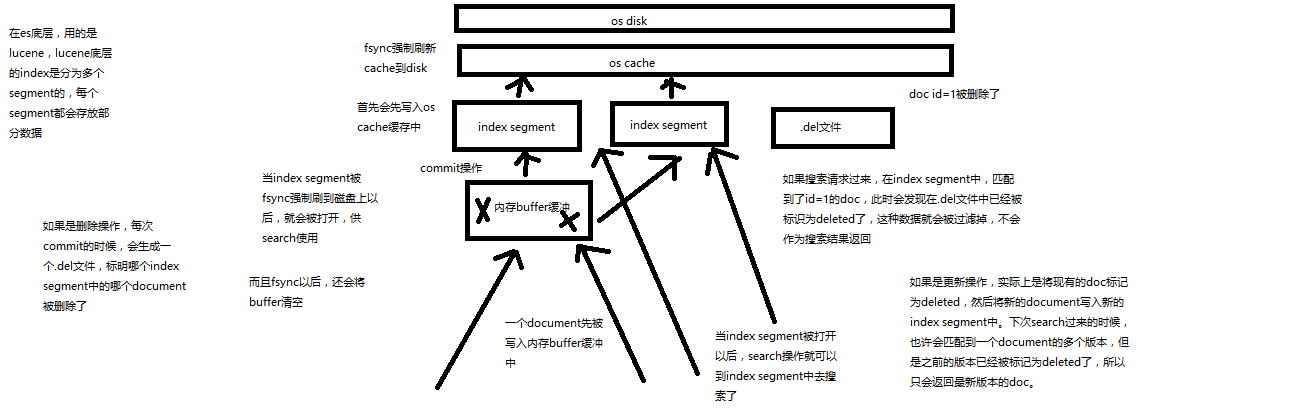
（4）等待在os cache中的index segment被fsync强制刷到磁盘上

（5）新的index sgement被打开，供search使用

（6）buffer被清空

每次commit point时，会有一个.del文件，标记了哪些segment中的哪些document被标记为deleted了

搜索的时候，会依次查询所有的segment，从旧的到新的，比如被修改过的document，在旧的segment中，会标记为deleted，在新的segment中会有其新的数据



## 内核原理探秘\_优化写入流程实现NRT近实时（filesystem cache，refresh）

现有流程的问题，每次都必须等待fsync将segment刷入磁盘，才能将segment打开供search使用，这样的话，从一个document写入，到它可以被搜索，可能会超过1分钟，这就不是近实时的搜索了。主要瓶颈在于fsync实际发生磁盘IO写数据进磁盘，是很耗时的。

写入流程别改进如下：

（1）数据写入buffer

（2）每隔一定时间，buffer中的数据被写入segment文件，但是先写入os cache

（3）只要segment写入os cache，那就直接打开供search使用，不立即执行commit

数据写入os cache，并被打开供搜索的过程，叫做refresh，默认是每隔1秒refresh一次。也就是说，每隔一秒就会将buffer中的数据写入一个新的index segment file，先写入os cache中。所以，es是近实时的，数据写入到可以被搜索，默认是1秒。

POST /my\_index/\_refresh，可以手动refresh，一般不需要手动执行，没必要，让es自己搞就可以了

比如说，我们现在的时效性要求，比较低，只要求一条数据写入es，一分钟以后才让我们搜索到就可以了，那么就可以调整refresh interval

PUT /my\_index

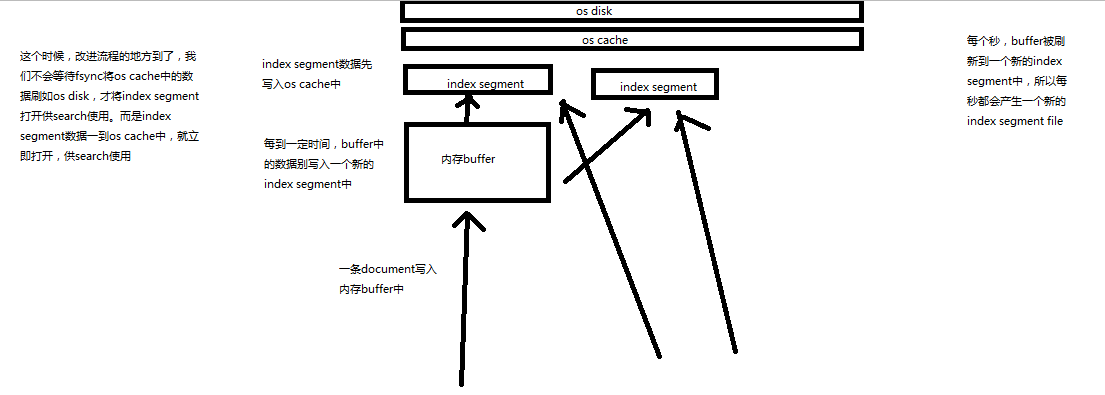
{

"settings": {

"refresh\_interval": "30s"

}

}



## 内核原理探秘\_继续优化写入流程实现durability可靠存储（translog，flush）

再次优化的写入流程

（1）数据写入buffer缓冲和translog日志文件

（2）每隔一秒钟，buffer中的数据被写入新的segment file，并进入os cache，此时segment被打开并供search使用

（3）buffer被清空

（4）重复1~3，新的segment不断添加，buffer不断被清空，而translog中的数据不断累加

（5）当translog长度达到一定程度的时候，commit操作发生

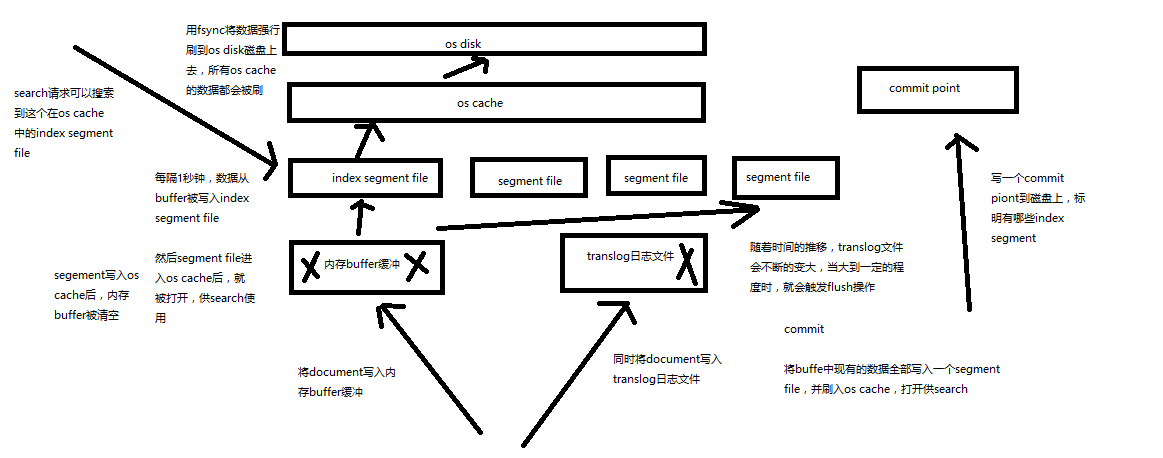
（5-1）buffer中的所有数据写入一个新的segment，并写入os cache，打开供使用

（5-2）buffer被清空

（5-3）一个commit ponit被写入磁盘，标明了所有的index segment

（5-4）filesystem cache中的所有index segment file缓存数据，被fsync强行刷到磁盘上

（5-5）现有的translog被清空，创建一个新的translog



基于translog和commit point，如何进行数据恢复



fsync+清空translog，就是flush，默认每隔30分钟flush一次，或者当translog过大的时候，也会flush

POST /my\_index/\_flush，一般来说别手动flush，让它自动执行就可以了

translog，每隔5秒被fsync一次到磁盘上。在一次增删改操作之后，当fsync在primary shard和replica shard都成功之后，那次增删改操作才会成功

但是这种在一次增删改时强行fsync translog可能会导致部分操作比较耗时，也可以允许部分数据丢失，设置异步fsync translog

PUT /my\_index/\_settings

{

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

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

}

## 内核原理探秘\_最后优化写入流程实现海量磁盘文件合并（segment merge，optimize）

每秒一个segment file，文件过多，而且每次search都要搜索所有的segment，很耗时

默认会在后台执行segment merge操作，在merge的时候，被标记为deleted的document也会被彻底物理删除

每次merge操作的执行流程

（1）选择一些有相似大小的segment，merge成一个大的segment

（2）将新的segment flush到磁盘上去

（3）写一个新的commit point，包括了新的segment，并且排除旧的那些segment

（4）将新的segment打开供搜索

（5）将旧的segment删除

POST /my\_index/\_optimize?max\_num\_segments=1，尽量不要手动执行，让它自动默认执行就可以了

