1. 简介
   1. 概述

Elasticsearch，分布式，高性能，高可用，可伸缩的搜索和分析系统

每个es节点都封装了lucene

<什么是Elasticsearch.png>

* 1. [Elasticsearch的功能、适用场景以及特点介绍](Elasticsearch的功能、适用场景以及特点介绍.txt)
  2. [Elasticsearch核心概念：NRT、索引、分片、副本等](Elasticsearch核心概念：NRT、索引、分片、副本等.txt)

1. 启动

<http://localhost:9200/?pretty来验证是否启动成功>

<http://localhost:5601>

使用kibana来开发

1. 简单的集群管理

es提供了一套api，叫做cat api，可以查看es中各种各样的数据

GET /\_cat/health?v

如何快速了解集群的健康状况？green、yellow、red？

green：每个索引的primary shard和replica shard都是active状态的

yellow：每个索引的primary shard都是active状态的，但是部分replica shard不是active状态，处于不可用的状态

red：不是所有索引的primary shard都是active状态的，部分索引有数据丢失了

为什么现在会处于一个yellow状态？

我们现在就一个笔记本电脑，就启动了一个es进程，相当于就只有一个node。现在es中有一个index，就是kibana自己内置建立的index。由于默认的配置是给每个index分配5个primary shard和5个replica shard，而且primary shard和replica shard不能在同一台机器上（为了容错）。现在kibana自己建立的index是1个primary shard和1个replica shard。当前就一个node，所以只有1个primary shard被分配了和启动了，但是一个replica shard没有第二台机器去启动。

做一个小实验：此时只要启动第二个es进程，就会在es集群中有2个node，然后那1个replica shard就会自动分配过去，然后cluster status就会变成green状态。

举例：单node环境下，创建一个index，有3个primary shard，3个replica shard，集群status是yellow。这个时候，只会将3个primary shard分配到仅有的一个node上去，另外3个replica shard是无法分配的。集群可以正常工作，但是一旦出现节点宕机，数据全部丢失，而且集群不可用，无法承接任何请求。

1. 索引增删

增：PUT /test\_index?pretty 删：DELETE /test\_index?pretty

然后通过GET /\_cat/health?v来检查

1. CRUD

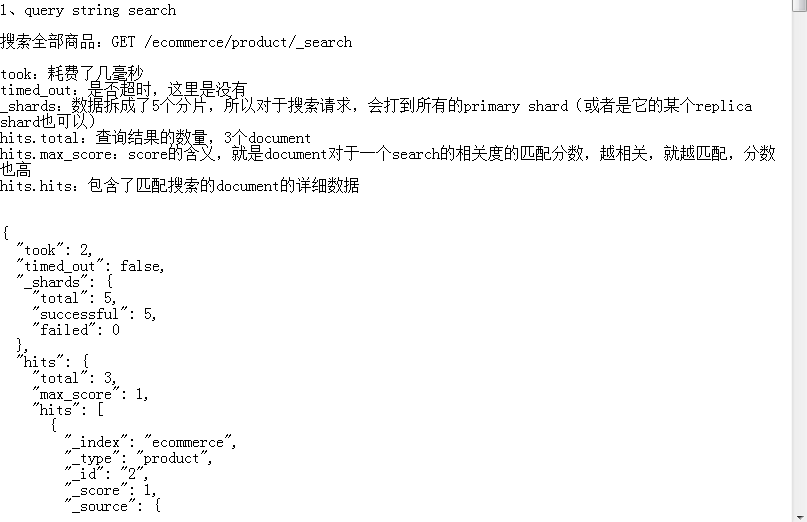
es会自动建立index和type，不需要提前创建，而且es默认会对document每个field都建立倒排索引，让其可以被搜索

* 1. 新增：PUT /index名/type名/id{...}
  2. 查询：
     1. GET /index名/type名/id{...}
     2. GET /search{}
     3. GET /index1,index2/type1,type2/search{}
     4. GET /\_search{ "from": 0, "size": 10}
  3. 更新：
     1. 替换PUT /index名/type名/id{...}

替换方式有一个不好，即使必须带上所有的field，才能去进行信息的修改，否则会把原有的数据给清掉。

* + 1. 修改POST /index名/type名/id/\_update{“doc”:{...}}
  1. 删除：DELETE /index名/type名/id

1. 多种搜索方式
   1. query string search



搜索商品名称中包含yagao的商品，而且按照售价降序排序：

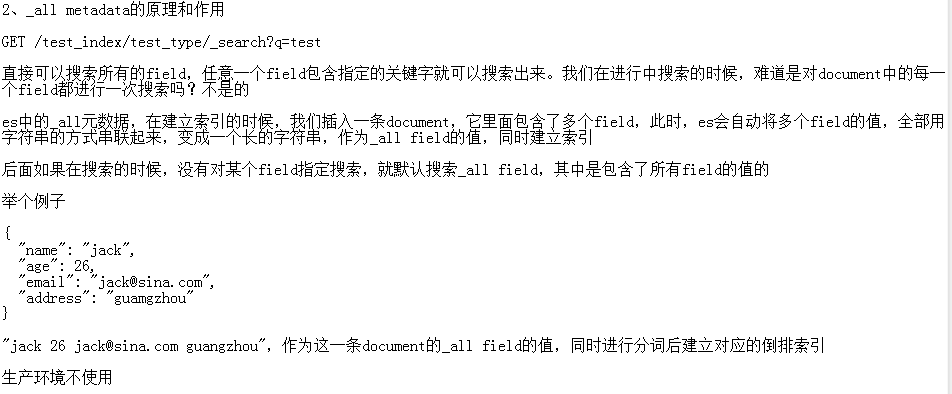
GET /ecommerce/product/\_search?q=name:yagao&sort=price:desc

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

GET /test\_index/test\_type/\_search?q=+test\_field:test

GET /test\_index/test\_type/\_search?q=-test\_field:test

-表示不包含

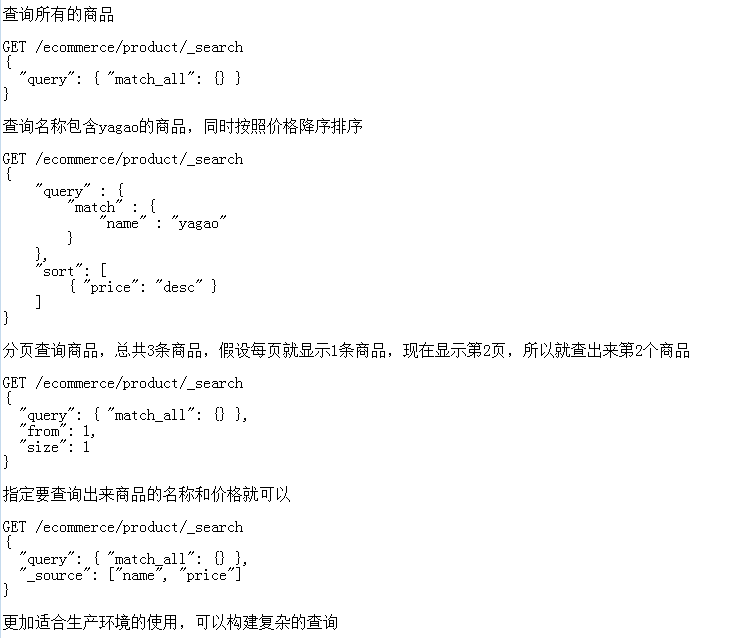


适用于临时的在命令行使用一些工具，比如curl，快速的发出请求，来检索想要的信息；但是如果查询请求很复杂，是很难去构建的.在生产环境中，几乎很少使用query string search

* 1. query DSL

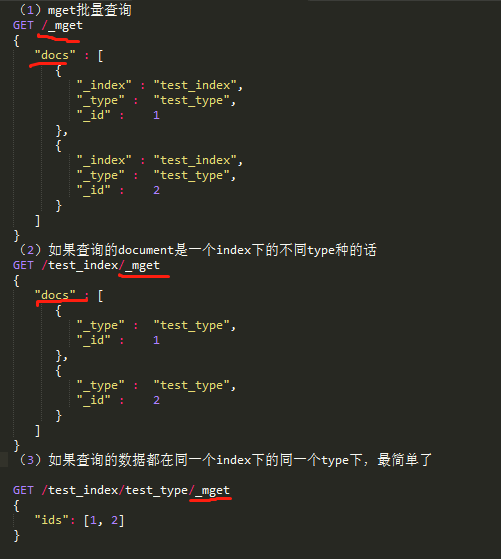
DSL：Domain Specified Language，特定领域的语言

http request body：请求体，可以用json的格式来构建查询语法，比较方便，可以构建各种复杂的语法，比query string search肯定强大多了

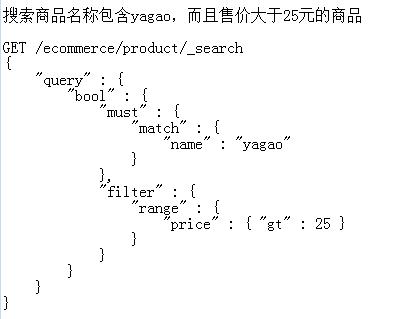


<图解partial update实现原理以及其优点.png>

* 1. 批量查询

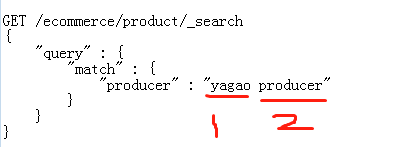


* 1. query filter

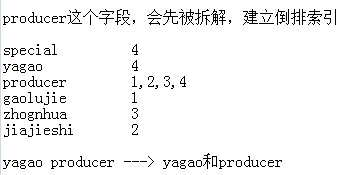


bool下面可以放must,must not,should,filter

* 1. full-text search



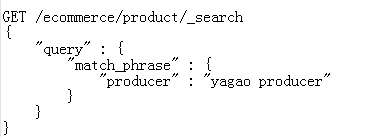
这里有两个词yagao和producer.所以是全文检索。可以看到查出的相关度分数是不一样的。



* 1. phrase search（短语搜索）

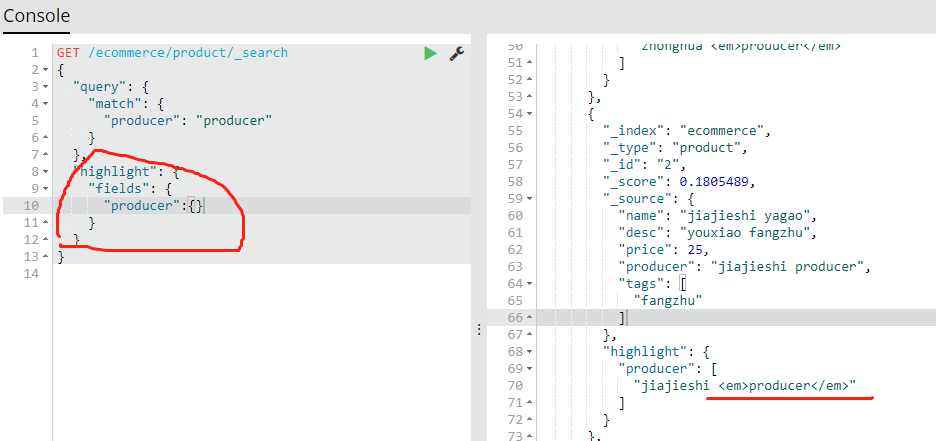
跟全文检索相对应，相反，全文检索会将输入的搜索串拆解开来，去倒排索引里面去一一匹配，只要能匹配上任意一个拆解后的单词，就可以作为结果返回

phrase search，要求输入的搜索串，必须在指定的字段文本中，完全包含一模一样的，才可以算匹配，才能作为结果返回

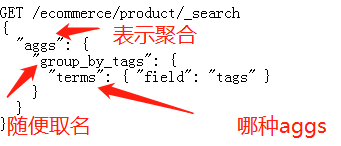


ES支持两种：精确搜索和全文搜索，精确就是==，full text就不是说单纯的只是匹配完整的一个值，而是可以对值进行拆分词语后（分词）进行匹配，也可以通过缩写、时态、大小写、同义词等进行匹配

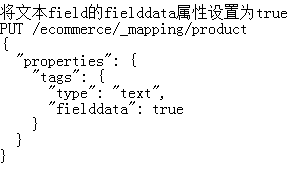
* 1. highlight search（高亮搜索结果）



* 1. 聚合分析



如果提示报错











* 1. bulk
     1. 语法

POST /\_bulk

{ "delete": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "3" }}

{ "create": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "12" }}

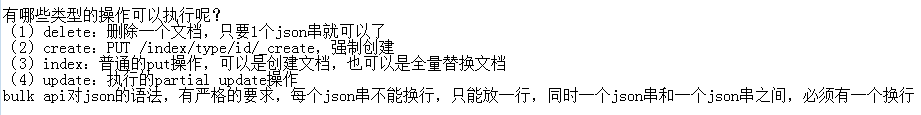
{ "test\_field": "test12" }

{ "index": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }



bulk操作中，任意一个操作失败，是不会影响其他的操作的，但是在返回结果里，会告诉你异常日志

* + 1. bulk size最佳大小

bulk request会加载到内存里，如果太大的话，性能反而会下降，因此需要反复尝试一个最佳的bulk size。一般从1000~5000条数据开始，尝试逐渐增加。另外，如果看大小的话，最好是在5~15MB之间。

* 1. multi-index和multi-type搜索模式

/\_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

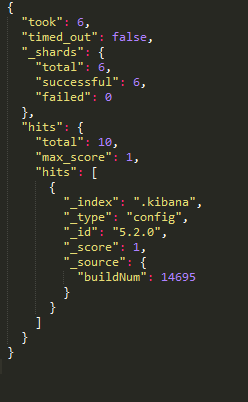
/index1/type1/\_search：搜索一个index下指定的type的数据

/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

1. 搜索出来的json 含义



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。

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

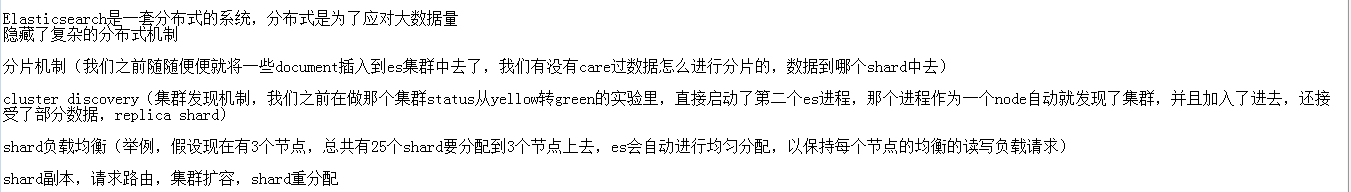
GET /\_search?timeout=10m

<timeout机制详解.png>

1. ES的基础分布式架构

<ES的基础分布式架构.png>

* 1. Elasticsearch对复杂分布式机制的透明隐藏特性



* 1. Elasticsearch的垂直扩容与水平扩容
  2. 增减或减少节点时的数据rebalance
  3. master节点
  4. 节点平等的分布式架构.节点对等，每个节点都能接收所有的请求,自动请求路由->响应收集
  5. shard&replica机制

（1）index包含多个shard

（2）每个shard都是一个最小工作单元，承载部分数据，lucene实例，完整的建立索引和处理请求的能力

（3）增减节点时，shard会自动在nodes中负载均衡

（4）primary shard和replica shard，每个document肯定只存在于某一个primary shard以及其对应的replica shard中，不可能存在于多个primary shard

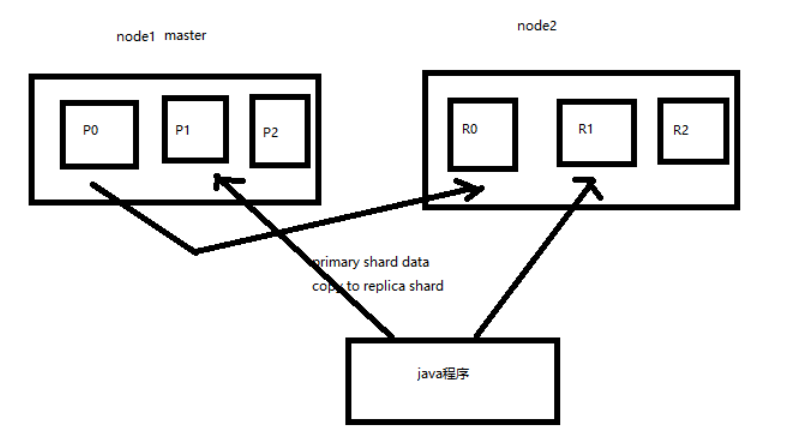
（5）replica shard是primary shard的副本，负责容错，以及承担读请求负载

（6）primary shard的数量在创建索引的时候就固定了，replica shard的数量可以随时修改

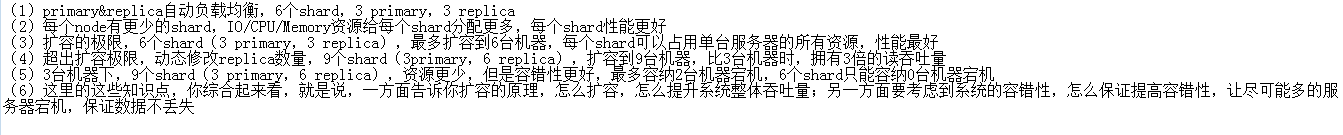
（7）primary shard的默认数量是5，replica默认是1，默认有10个shard，5个primary shard，5个replica shard

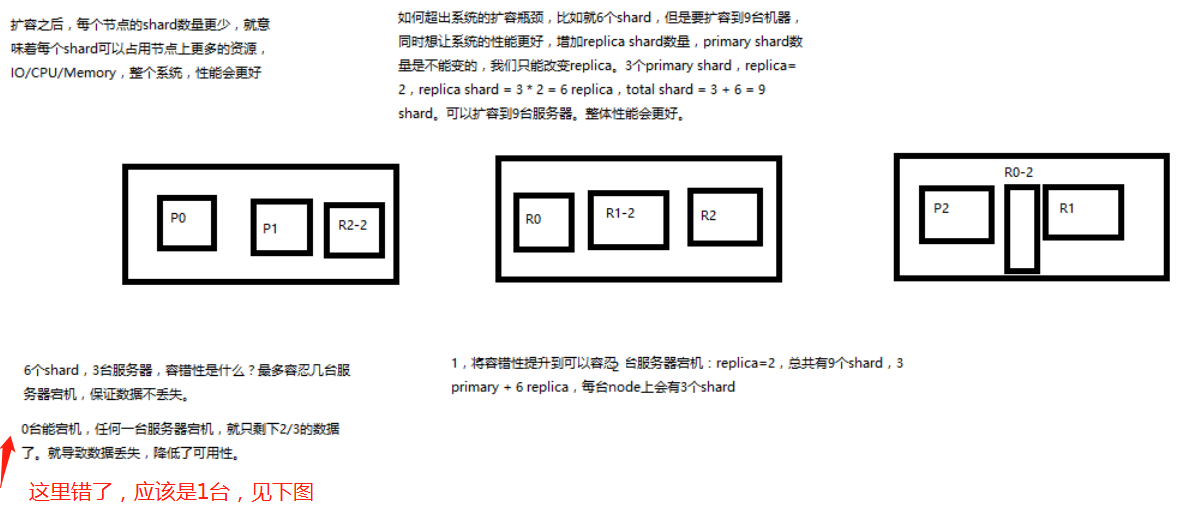
（8）primary shard不能和自己的replica shard放在同一个节点上（否则节点宕机，primary shard和副本都丢失，起不到容错的作用），但是可以和其他primary shard的replica shard放在同一个节点上

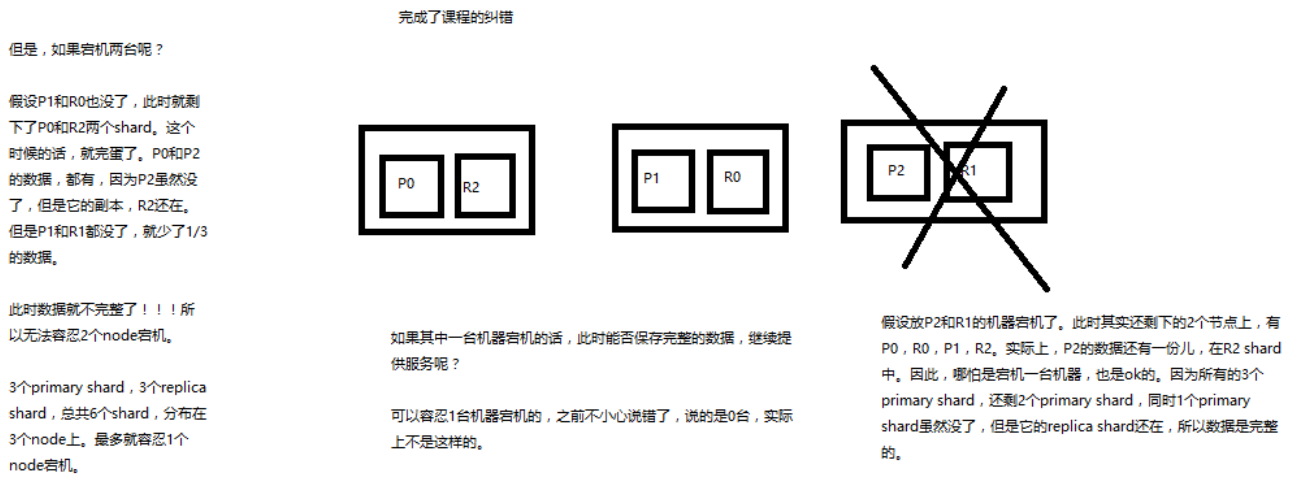
1. shard&replica机制
   1. 单node环境中创建index
   2. 2个node环境下replica shard如何分配



* + 1. replica shard分配：3个primary shard，3个replica shard，1 node
    2. primary ---> replica同步
    3. 读请求：primary/replica
  1. 横向扩容过程，如何超出扩容极限，以及如何提升容错性







* 1. ES容错过程分析

假设9 shard，3 node

master node宕机，自动master选举，red

replica容错：新master将replica提升为primary shard，yellow

重启宕机node，master copy replica到该node，使用原有的shard并同步宕机后的修改，green

<es容错过程分析.png>

1. document核心元数据
   1. 三种元数据
      1. \_index元数据
         1. 代表一个document存放在哪个index中
         2. 类似的数据放在一个索引，非类似的数据放不同索引：product index（包含了所有的商品），sales index（包含了所有的商品销售数据），inventory index（包含了所有库存相关的数据）。如果你把比如product，sales，human resource（employee），全都放在一个大的index里面，比如说company index，不合适的。
         3. index中包含了很多类似的document：类似是什么意思，其实指的就是说，这些document的fields很大一部分是相同的，你说你放了3个document，每个document的fields都完全不一样，这就不是类似了，就不太适合放到一个index里面去了。下图是反例，不应该这样创建index
         4. 索引名称必须是小写的，不能用下划线开头，不能包含逗号

<index如何创建的反例分析.png>

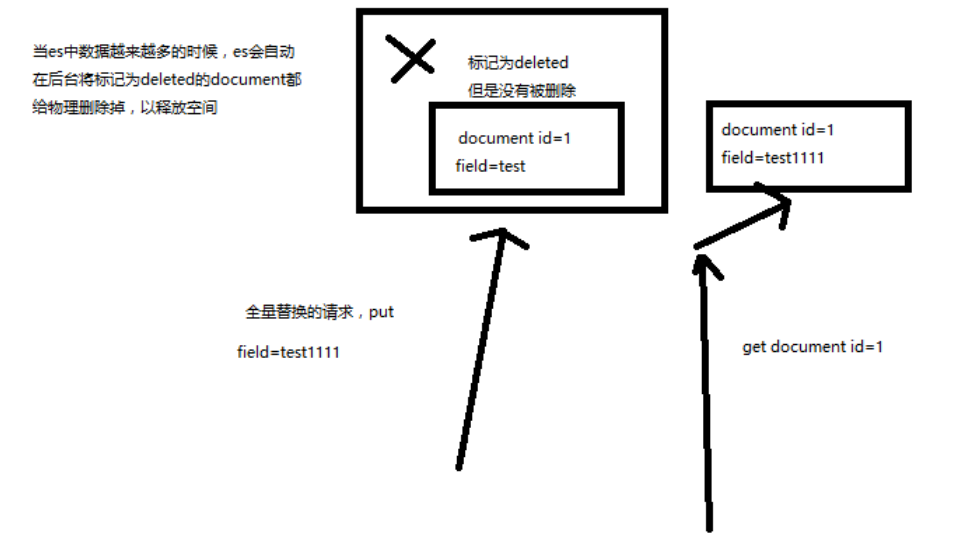
* + 1. \_type元数据
       1. 代表document属于index中的哪个类别（type）
       2. 一个索引通常会划分为多个type，逻辑上对index中有些许不同的几类数据进行分类：因为一批相同的数据，可能有很多相同的fields，但是还是可能会有一些轻微的不同，可能会有少数fields是不一样的，举个例子，就比如说，商品，可能划分为电子商品，生鲜商品，日化商品，等等。
       3. type名称可以是大写或者小写，但是同时不能用下划线开头，不能包含逗号
    2. \_id元数据

代表document的唯一标识，与index和type一起，可以唯一标识和定位一个document.我们可以手动指定document的id（put /index/type/id），也可以不指定，由es自动为我们创建一个id

* + 1. \_source元数据

使用的那个放在request body中的json串，默认情况下，在get的时候，会原封不动的给我们返回回来。如果不想返回那么多，可以在后面加?\_source=属性名或者使用上面查询的querydsl方式

* 1. document id的手动指定与自动生成两种方式
  2. document的全量替换、强制创建以及图解lazy delete机制



* + 1. document的全量替换

document是不可变的，如果要修改document的内容，第一种方式就是全量替换，直接对document重新建立索引，替换里面所有的内容。es会将老的document标记为deleted，然后新增我们给定的一个document，当我们创建越来越多的document的时候，es会在适当的时机在后台自动删除标记为deleted的document

* + 1. document的强制创建

PUT /index/type/id?op\_type=create或者PUT /index/type/id/\_create，一般用后者

* + 1. document的删除

DELETE /index/type/id，不会理解物理删除，只会将其标记为deleted，当数据越来越多的时候，在后台自动删除

1. 并发
   1. 并发冲突

<深度图解剖析Elasticsearch并发冲突问题.png>

* 1. 悲观锁与乐观锁两种并发控制方案

<深度图解剖析悲观锁与乐观锁两种并发控制方案.png>

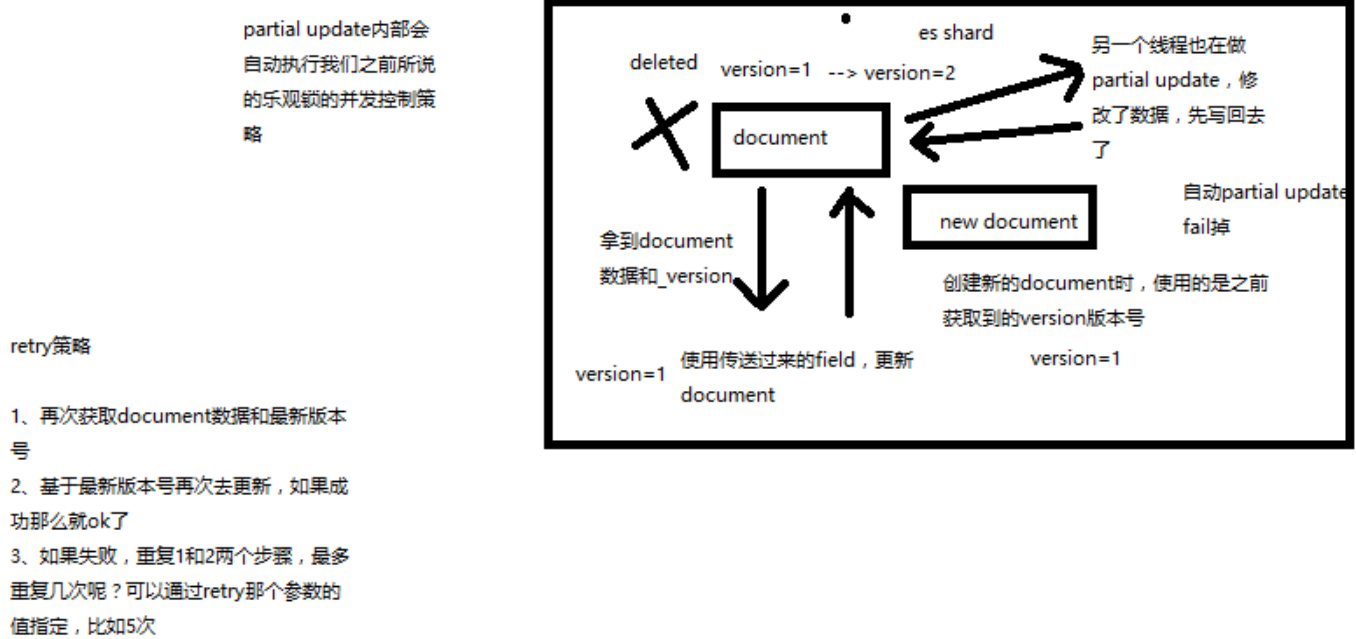
* 1. 基于\_version进行乐观锁并发控制

第一次创建一个document的时候，它的\_version内部版本号就是1；以后，每次对这个document执行修改或者删除操作，都会对这个\_version版本号自动加1；哪怕是删除，也会对这条数据的版本号加1

<图解Elasticsearch内部如何基于_version进行乐观锁并发控制.png>

partial update乐观锁并发控制原理以及相关操作

可以通过post /index/type/id/\_update?retry\_on\_conflict=5&version=6来定义重试次数



* + 1. 基于\_version进行乐观锁并发控制
       1. 先构造一条数据出来

PUT /test\_index/test\_type/7{ "test\_field": "test test"}

* + - 1. 模拟两个客户端，都获取到了同一条数据

GET test\_index/test\_type/7

* + - 1. 其中一个客户端，先更新了一下这个数据，同时带上数据的版本号，确保说，es中的数据的版本号，跟客户端中的数据的版本号是相同的，才能修改

PUT /test\_index/test\_type/7?version=1 { "test\_field": "test client 1"}

* + - 1. 另外一个客户端，尝试基于version=1的数据去进行修改，同样带上version版本号，进行乐观锁的并发控制

PUT /test\_index/test\_type/7?version=1 {"test\_field": "test client 2"}

* + - 1. 在乐观锁成功阻止并发问题之后，尝试正确的完成更新，基于最新的数据和版本号，去进行修改，修改后，带上最新的版本号，可能这个步骤会需要反复执行好几次，才能成功，特别是在多线程并发更新同一条数据很频繁的情况下

GET /test\_index/test\_type/7

PUT /test\_index/test\_type/7?version=2 { "test\_field": "test client 2"}

* + 1. 基于external version进行乐观锁并发控制

?version=1&version\_type=external唯一的区别在于，\_version，只有当你提供的version与es中的\_version一模一样的时候，才可以进行修改，只要不一样，就报错；当version\_type=external的时候，只有当你提供的version比es中的\_version大的时候，才能完成修改

1. document数据路由原理

<document路由原理.png>

* 1. 算法

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，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。0,1,2。

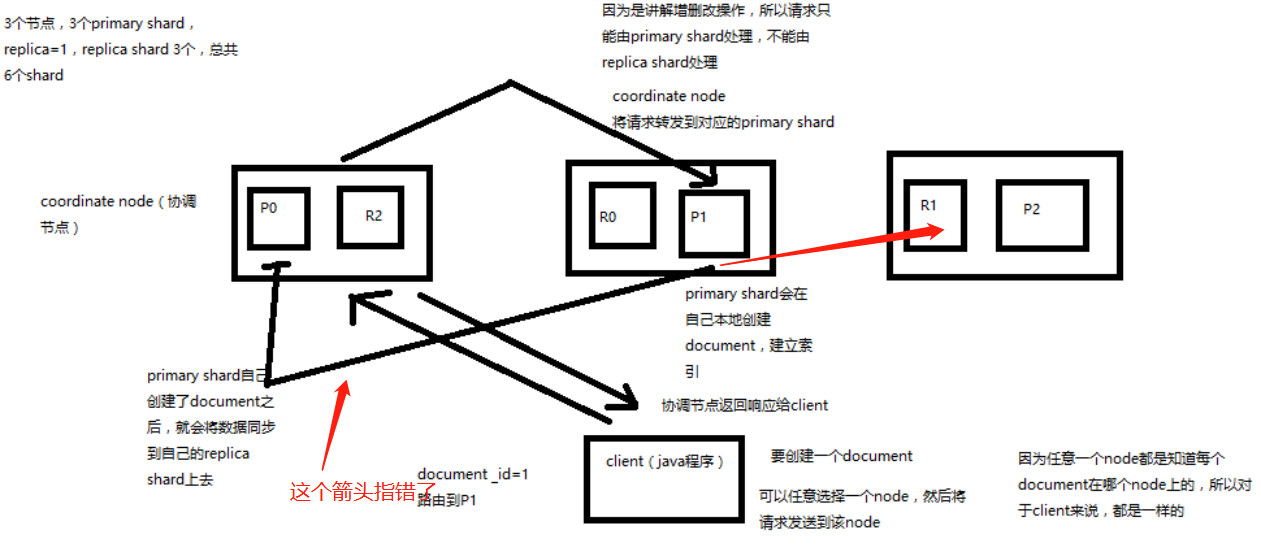
所以说primary shard一旦index建立，是不允许修改的。（见图）。但是replica shard是可以随时修改。

* 1. \_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。

1. document增删改内部原理



（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

1. document读请求内部原理

<读请求内部原理.png>

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

1. 写一致性原理以及quorum机制
   1. consistency，one（primary shard），all（all shard），quorum（default）

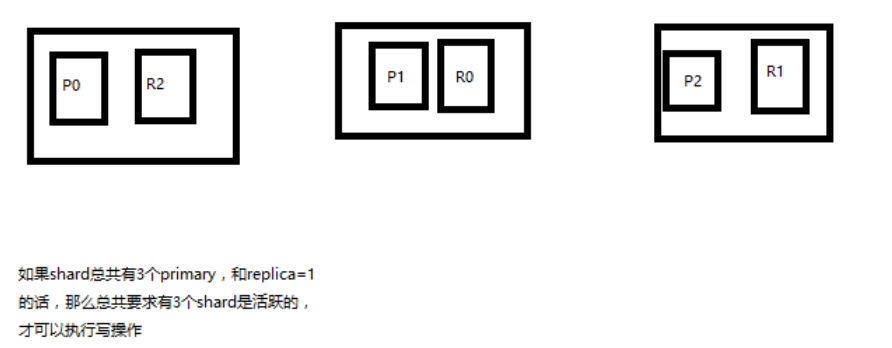
我们在发送任何一个增删改操作的时候，比如说put /index/type/id，都可以带上一个consistency参数，指明我们想要的写一致性是什么？

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（见下图）



* 1. 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

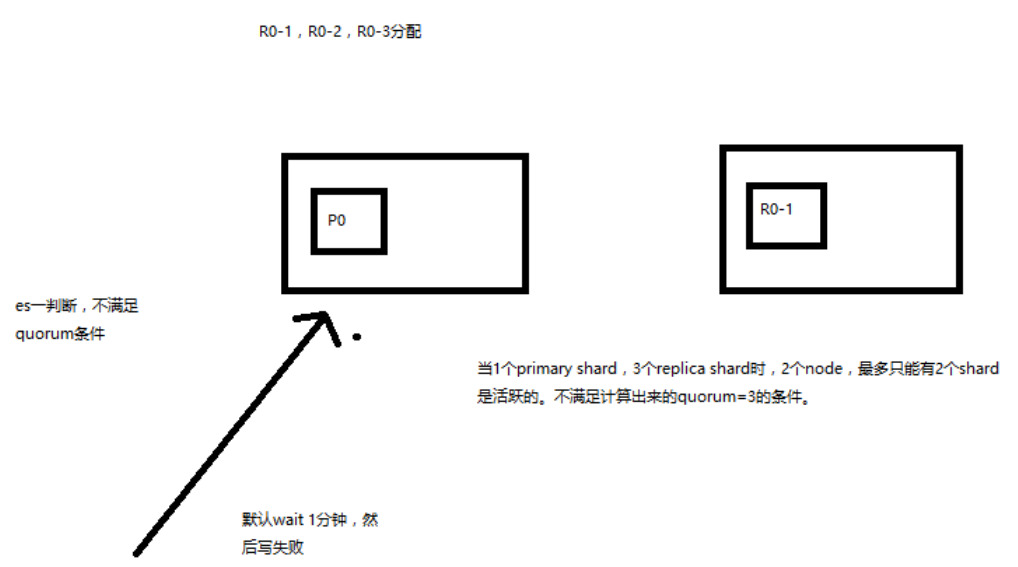
quorum = int( (3 + 1) / 2 ) + 1 = 3

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

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

1个primary shard，replica=3，quorum=((1 + 3) / 2) + 1 = 3，要求1个primary shard + 3个replica shard = 4个shard，其中必须有3个shard是要处于active状态的。如果这个时候只有2台机器的话，会出现什么情况呢？见下图



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

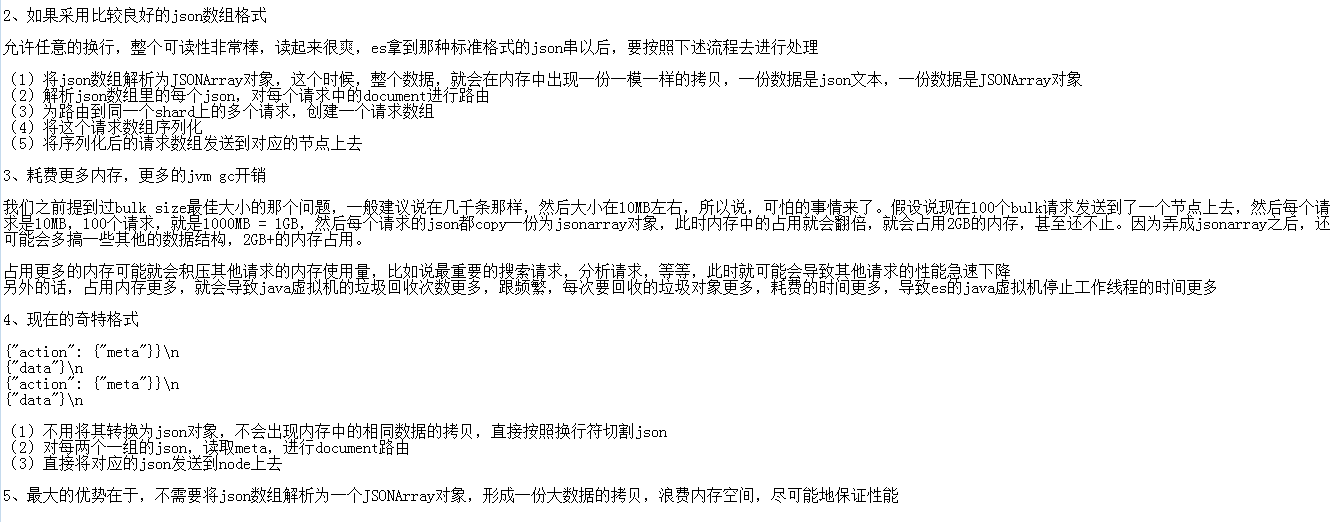
(1 + 1 / 2) + 1 = 2，要求必须有2个shard是活跃的，但是可能就1个node，此时就1个shard是活跃的，如果你不特殊处理的话，导致我们的单节点集群就无法工作

quorum不齐全时，wait，默认1分钟，timeout，100，30s

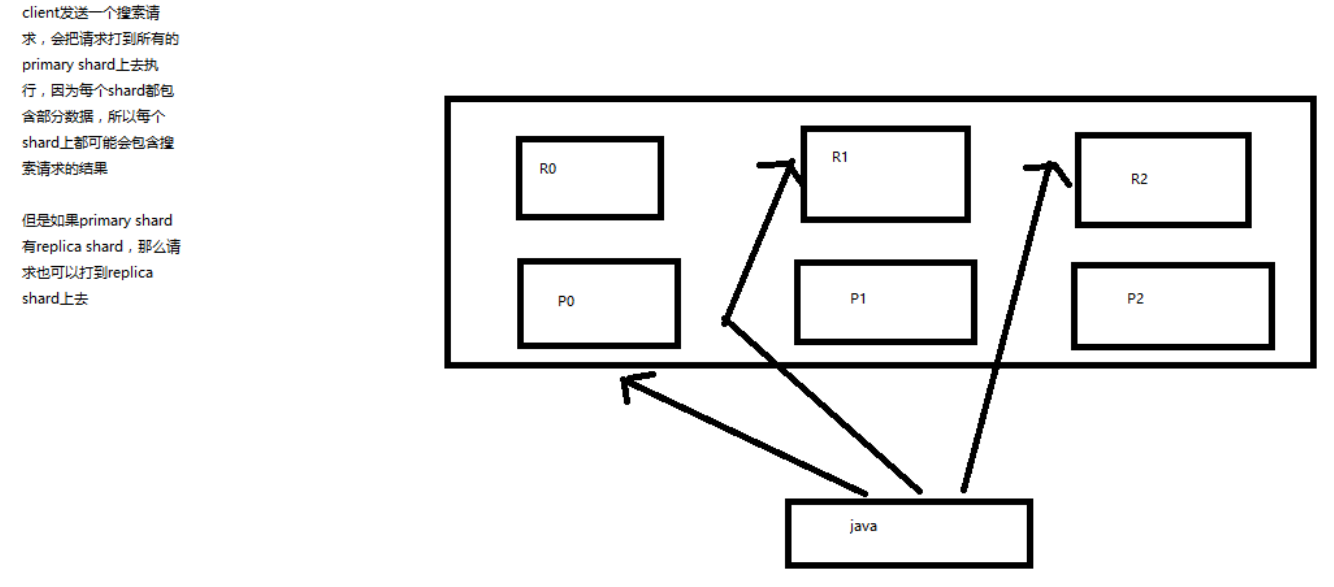
等待期间，期望活跃的shard数量可以增加，最后实在不行，就会timeout

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

1. bulk api的奇特json格式与底层性能优化关系



1. 搜索原理



[fetch phase原理](fetch phase.png)

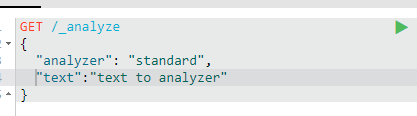
1. deep paging

<deep paging图解.png>

1. 倒排索引

拆分--》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)

1. 分词器
   1. 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），& --> and（I&you --> I and you）
   2. tokenizer：分词，hello you and me --> hello, you, and, me
   3. token filter：lowercase，stop word，synonymom，dogs --> dog，liked --> like，Tom --> tom，a/the/an --> 干掉，mother --> mom，small --> little
   4. 内置分词器
      1. standard analyzer：（默认的是standard）
      2. simple analyzer
      3. whitespace analyzer
      4. language analyzer（特定的语言的分词器，比如说，english，英语分词器）
   5. 测试分词器



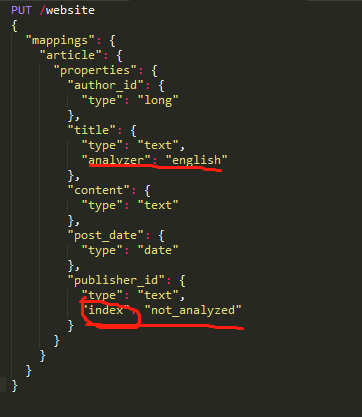
1. mapping
   1. 概念：

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。也可以手动在创建数据之前，先创建index和type，以及type对应的mapping。

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。

* 1. 手动建立和修改mapping以及定制string类型数据是否分词



1. bouncing results问题解决方案

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

1. 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scoll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

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

流程：先GET

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

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

"sort": [ "\_doc" ], "size": 3}

会得到结果，结果中有个scrollid,下次查询把这个scrollid给带上

GET /\_search/scroll

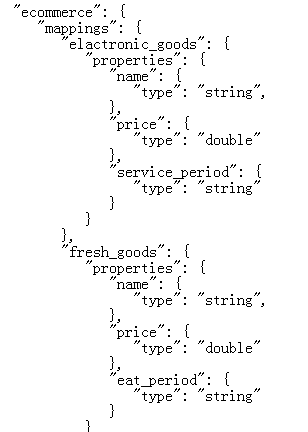
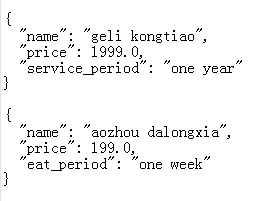
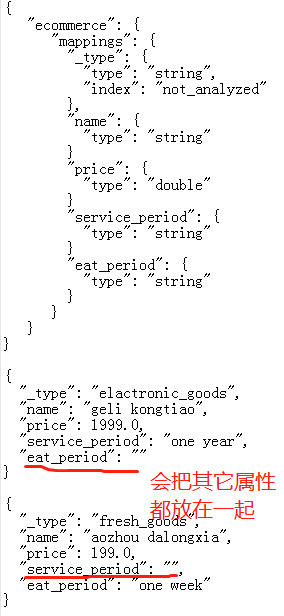
{"scroll":"1m","scroll\_id" "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACxeFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAALF8WNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACxhFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYhY0b25zVFlWWlRqR3ZJajlfc3BXejJ3"

}

1. 索引管理
   1. type底层数据结构：

可以通过GET /索引名/\_mapping/type名来查看

mapping +数据------------------------------------>底层结构

+--->

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

* 1. \_mapping 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）

* + 1. properties :type:类型 index:要不要分词 analyzer:使用哪个分词器
    2. \_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

* + 1. \_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。

* + 1. 标识性metadata:\_index，\_type，\_id
  1. 手动增删改mapping的索引



* 1. 定制化自己的dynamic mapping策略
     1. dynamic策略

true：遇到陌生字段，就进行dynamic mapping

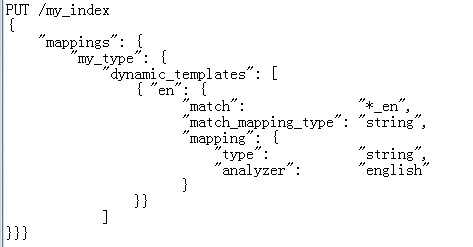
false：遇到陌生字段，就忽略

strict：遇到陌生字段，就报错

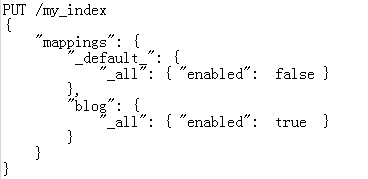
* + 1. dynamic mapping策略
       1. date\_detection

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。

* + - 1. 定制自己的dynamic mapping template（type level）



* + - 1. 定制自己的default mapping template（index level）



1. [基于scoll+bulk+索引别名实现零停机重建索引.](基于scoll+bulk+索引别名实现零停机重建索引.txt)
2. 内核原理
   1. 倒排索引组成结构以及其索引可变原因
   2. document写入原理（buffer，segment，commit）（并不是ES最终流程，只是一个很粗的流程，后面几点ES做了优化，最终实现了ES的最终流程）
      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
      7. 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

[document写入原理图解](document增删改内核级原理.png)

* 1. 优化写入流程实现NRT近实时（filesystem cache，refresh）
     1. 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。
     2. 优化前面的第5点，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" }}

[优化-NRT近实时图解](NRT写入优化.png)

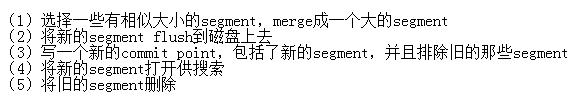
* 1. 继续优化写入流程实现durability可靠存储（translog，flush）（ES最终流程）
     1. 数据写入buffer缓冲和translog日志文件
     2. 每隔一秒钟，buffer中的数据被写入新的segment file，并进入os cache，此时segment被打开并供search使用
     3. buffer被清空
     4. 重复1~3，新的segment不断添加，buffer不断被清空，而translog中的数据不断累加
     5. 当translog长度达到一定程度的时候，commit操作发生
        1. buffer中的所有数据写入一个新的segment，并写入os cache，打开供使用
        2. buffer被清空
        3. 一个commit ponit被写入磁盘，标明了所有的index segment
        4. filesystem cache中的所有index segment file缓存数据，被fsync强行刷到磁盘上
        5. 现有的translog被清空，创建一个新的translog

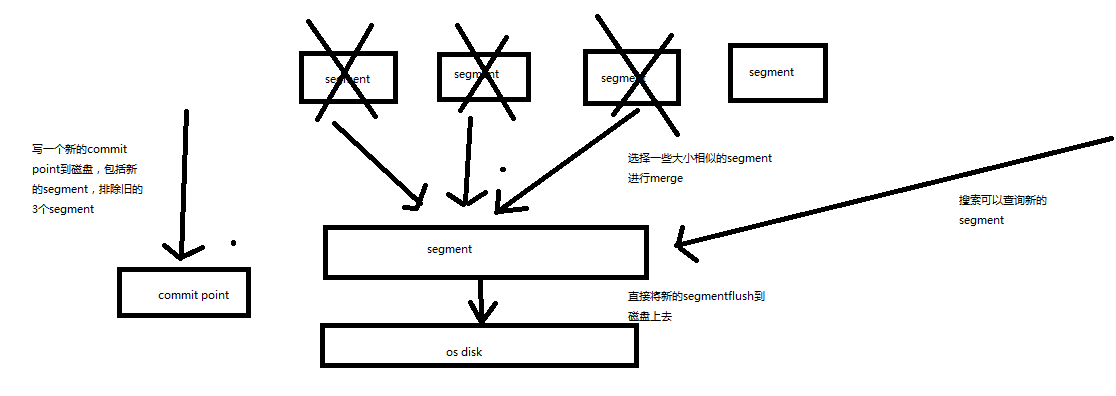
[终极版本的es数据写入流程](终极版本的es数据写入流程.png)

* 1. 基于translog和commit point，如何进行数据恢复
     1. 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
     2. 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
     3. 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

[commit point+translog进行数据恢复](commit point+translog进行数据恢复.png)

* 1. 最后优化写入流程实现海量磁盘文件合并（segment merge，optimize）
     1. 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. 与JAVA整合
   1. 如果插入数据直接查询的话是有可能搜索不到的。近实时！！！默认是1秒以后，写入es的数据，才能被搜索到。需要停1秒才可以
   2. 案例

[CRUD](EmployeeCRUDApp.java)

SEARCH：

SearchResponse searchResponse = client.prepareSearch("company").setTypes("employee")  
 .setQuery(QueryBuilders.*matchQuery*("position", "technique"))  
 .setPostFilter(QueryBuilders.*rangeQuery*("age").from(30).to(40))  
 .setFrom(0).setSize(1).get();

聚合：首先按照country国家来进行分组，然后在每个country分组内，再按照入职年限进行分组，最后计算每个分组内的平均薪资

SearchResponse searchResponse = client.prepareSearch("company")   
 .addAggregation(AggregationBuilders.*terms*("group\_by\_country").field("country")  
 .subAggregation(AggregationBuilders  
 .*dateHistogram*("group\_by\_join\_date")  
 .field("join\_date")  
 .dateHistogramInterval(DateHistogramInterval.*YEAR*)  
 .subAggregation(AggregationBuilders.*avg*("avg\_salary").field("salary")))  
 )  
 .execute().actionGet();  
  
Map<String, Aggregation> aggrMap = searchResponse.getAggregations().asMap();  
  
StringTerms groupByCountry = (StringTerms) aggrMap.get("group\_by\_country");  
Iterator<Bucket> groupByCountryBucketIterator = groupByCountry.getBuckets().iterator();  
while(groupByCountryBucketIterator.hasNext()) {  
 Bucket groupByCountryBucket = groupByCountryBucketIterator.next();  
 System.*out*.println(groupByCountryBucket.getKey() + ":" + groupByCountryBucket.getDocCount());   
  
 Histogram groupByJoinDate = (Histogram) groupByCountryBucket.getAggregations().asMap().get("group\_by\_join\_date");  
 Iterator<Histogram.Bucket> groupByJoinDateBucketIterator = groupByJoinDate.getBuckets().iterator();  
 while(groupByJoinDateBucketIterator.hasNext()) {  
 Histogram.Bucket groupByJoinDateBucket = groupByJoinDateBucketIterator.next();  
 System.*out*.println(groupByJoinDateBucket.getKey() + ":" +groupByJoinDateBucket.getDocCount());   
   
 Avg avg = (Avg) groupByJoinDateBucket.getAggregations().asMap().get("avg\_salary");   
 System.*out*.println(avg.getValue());   
 }  
}