# 1、elasticsearch安装搭建以及基本原理

搜索引擎包括全文索引（百度，谷歌）、目录索引、元搜索引擎、垂直搜索引擎（电商类的，专业领域类等）、集合式搜索引擎、门户搜索引擎与免费链接列表等，公司内部使用垂直搜索

## es安装与基本原理

### es特性

#### 1.1 ES 与 Solr 对比：

当单纯的对已有数据进行搜索时，Solr更快。

当实时建立索引时, Solr会产生io阻塞，查询性能较差, Elasticsearch具有明显的

优势。

大型互联网公司，实际生产环境测试，将搜索引擎从Solr转到 Elasticsearch以后

的平均查询速度有了50倍的提升。

**总结：**

二者安装都很简单。

1、Solr 利用 Zookeeper 进行分布式管理，而Elasticsearch 自身带有分布式协调

管理功能。

2、Solr 支持更多格式的数据，比如JSON、XML、CSV，而 Elasticsearch 仅支持

json文件格式。

3、Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时

效率明显低于 Elasticsearch。

4、Solr 是传统搜索应用的有力解决方案，但 Elasticsearch更适用于新兴的实时

搜索应用。

倒排索引：由关键词对应一系列文章，及关键词匹配文档，而正向索引是在每行数据中去找对应的属性值

#### 1.2 es的文档映射

1.ES中映射可以分为动态映射和静态映射

动态映射：

在关系数据库中，需要事先创建数据库，然后在该数据库下创建数据表，并创建

表字段、类型、长度、主键等，最后才能基于表插入数据。而Elasticsearch中不

需要定义Mapping映射（即关系型数据库的表、字段等），在文档写入

Elasticsearch时，会根据文档字段自动识别类型，这种机制称之为动态映射。

动态映射规则如下：



静态映射：

静态映射是在Elasticsearch中也可以事先定义好映射，包含文档的各字段类

型、分词器等，这种方式称之为静态映射。

#### 1.3 es的并发控制

在数据库领域中，有两种方法来确保并发更新，不会丢失数据：

1、悲观并发控制

这种方法被关系型数据库广泛使用，它假定有变更冲突可能发生，因此阻塞访问

资源以防止冲突。 一个典型的例子是读取一行数据之前先将其锁住，确保只有放

置锁的线程能够对这行数据进行修改。

2、乐观并发控制

Elasticsearch 中使用的这种方法假定冲突是不可能发生的，并且不会阻塞正在

尝试的操作。然而，如果源数据在读写当中被修改，更新将会失败。应用程序接

下来将决定该如何解决冲突。例如，可以重试更新、使用新的数据、或者将相关

情况报告给用户。

对文档执行的每个操作都会由协调该更改的主分片分配一个序列号。每次操作都会增加序列号，因此新操作可以保证比旧操作具有更高的序列号。然后，Elasticsearch 可以使用操作的序列号来确保更新的文档版本永远不会被分配了较小序列号的更改覆盖。所有这些操作都是es自动控制，不用程序处理，不过需要做重试逻辑，当因为版本号返回异常时需要重新查一下数据返回最新的版本号，然后重试修改操作

使用version控制并发，version为es内部实现，程序中直接使用即可

PUT /db\_index/\_doc/1?version=1{

"name": "Jack",

"sex": 1,

"age": 25,

"book": "Spring Boot 入门到精通",

"remark": "hello world"

}

ES新版本不使用version进行并发版本控制 if\_seq\_no=版本值

&if\_primary\_term=文档位置

\_seq\_no：文档版本号，作用同\_version

\_primary\_term：文档所在位置,即主分片的编号

POST /my\_doc/\_update/1/?if\_seq\_no=5&if\_primary\_term=1{

"doc": {

"name": "kevin"

}}

**if\_seq\_no 和 if\_primary\_term** 是用来并发控制，他们和version不同，version属于当个文档，而seq\_no属于整个index（每个文档会存储自己改变时的版本号，当再次修改时会对比当前文档的版本号，而不是索引最新的版本号），\_primary\_term主要是用来恢复数据时处理当多个文档的\_seq\_no一样时的冲突，比如当一个shard宕机了，raplica需要用到最新的数据，就会根据\_primary\_term和\_seq\_no这两个值来拿到最新的document

### 1.2 分布式索引概念介绍

1.number\_of\_shards：分片数量，类似于数据库里面分库分表，一经定义不可更改。主要响应写操作

2.number\_of\_replicas：副本数，用于备份分片的，和分片里面的数据保持一致，主要响应读操作，副本越多读取就越快，主分片挂掉后可以升级。

3.分布式索引一定要注意分片数量不能更改，所以在创建的时候一定要预先估算好数据大小，一般在8CPU 16G的机器上一个分片不要超过500g（不怕的话就500g，只要你索引数据结构优化的好，一般的是没问题）。索引会根据分片的配置来均匀的响应用户请求

4.如果调整了分片数那就要重建索引。

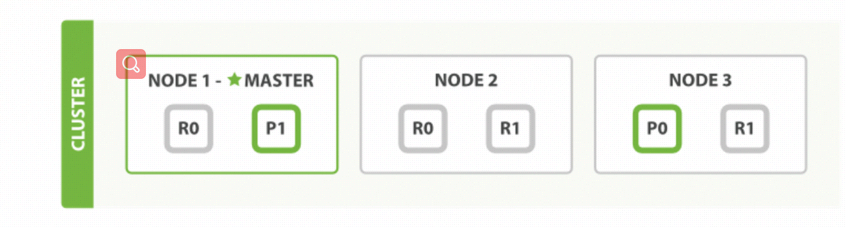
5.为什么shards不能修改：

因为我们采取的是分布式索引，假如我们有4篇文档id分别为：1，2，3，4。相信大家再数据库的分表中都知道有一个取模算法。用它来分配记录到对应的表中，其实es也采取的是这种思路。

所以13 % 2 ==> 1那么就会在第二个分片上。2 4 % 2 ==> 0那么就会在第一个分片上。

如果这时候你把分片数变了，很显然数据就不对了。所以分片数一旦变化需要重新全部建索引

6.索引写操作在集群上的操作：



假设我们集群如上面所示：这上面每个节点都可以接收请求，那么新建的一个流程如下所示：



以下是在主副分片和任何副本分片上面成功新建，索引和删除文档所需要的步骤顺序：

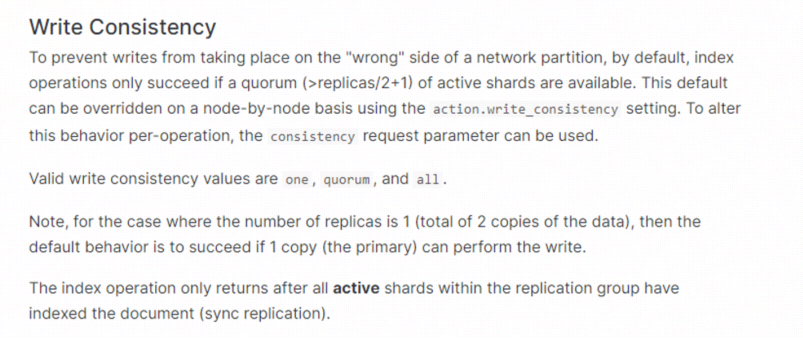
(1)客户端向Node1发送新建、索引或者删除请求。

(2)节点使用文档的\_id确定文档属于分 0。请求会被转发到Node3因为分片 0 的主分片目前被分配在Node3上。

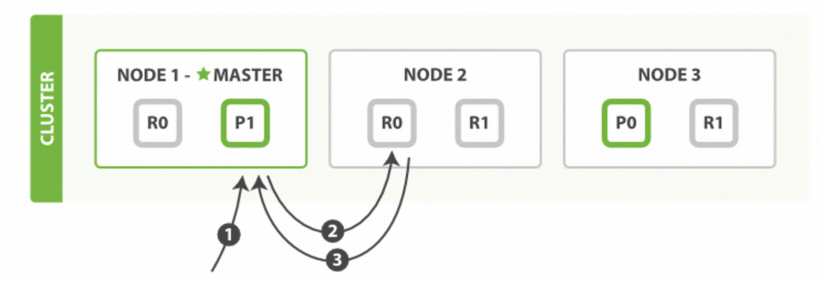
(3)Node 3在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1和 Node 2 的副本分片上。一旦所有的副本分片都报告成功, Node 3 将向协调节点报告成功，协调节点向客户端报告成功。

在客户端收到成功响应时，文档变更已经在主分片和所有副本分片执行完成，变更是安全的。

（这里课上讲的应该有点问题，现在的es版本是可用shard过半就可以写入，成功后会发送消息给master，我查了es最新的官方文档确实是这样的。所以如果书分片设置为1副本的时候，那默认的就是主分片写如成功了就会返回成功。）以下为es官方文档：



7、读操作



以下是从主分片或者副本分片检索文档的步骤顺序：

(1)客户端向Node1发送获取请求。

(2)节点使用文档的\_id来确定文档属于分片0,分片0的数据在三个节点上都有。 在这种情况下，它会根据负载均衡策略将请求转发到其中一个，比如Node2。

(3)Node2将文档返回给Node1然后将文档返回给客户端。

在处理读取请求时，协调结点在每次请求的时候都会通过轮询所有的副本分片来达到负载均衡，主分片也会接受请求，es有自适应副本功能，使用最合适的副本。

在文档被检索时，已经被索引的文档可能已经存在于主分片上但是还没有复制到副本分片。 在这种情况下，副本分片可能会报告文档不存在，但是主分片可能成功返回文档。

### 1.3 es的安装

### 1.4 es如何保证数据不丢失

在elasticsearch和磁盘之间还有一层cache也就是filesystem cache（通过内存buffer生成一个segment），大部分新增或者修改，删除的数据都在这层cache中，如果没有flush操作，那么就不能100%保证系统的数据不会丢失，比如突然断电或者机器宕机了，但实际情况是es中默认是30分钟才flush一次磁盘，这么长的时间内，如果发生不可控的故障，那么如何保证数据不丢失呢？

在es里面引入了transaction log（简称translog），这个log的作用就是每条数据的任何操作都会被记录到该log中，非常像Hadoop里面的edits log和hbase里面的WAL log

transaction log的工作流程如下：

（1）当一个文档被索引时，它会被添加到内存buffer里面同时也会在translog里面追加

（2）当每个shard每秒执行一次refresh操作完毕后，内存buffer会被清空但translog不会。

过程如下：

2.1 当refresh动作执行完毕后，内存buffer里面的数据会被写入到一个segment里面，这个还在cache中，并没有执行flush命令

2.2 新生成的segment在cache中，会被打开，这个时候就可以搜索新加的数据

2.3 最后内存buffer里面的数据会被清空

**segment合并**

Segment太多时，ES定期会将多个segment合并成为大的segment，减少索引查询时

IO开销，此阶段ES会真正的物理删除（之前执行过的delete的数据）

（3）随着更多的document添加，内存buffer区会不断的refresh，然后clear，但translog数量却越增越多

（4）当达到默认的30分钟时候，translog也会变得非常大，这个时候index要执行一次flush操作，同时会生成一个新的translog文件，并且要执行full commit操作，流程如下：

4.1 内存buffer里的所有document会被生成一个新的segment

4.2 然后segment被refresh到系统cache后，内存buffer会被清空

4.3 接着commit point会被写入到磁盘上

4.4 filesystem cache会被flush到磁盘上通过fsync操作

4.5 最后旧的translog会被删除，并会生成一个新的translog

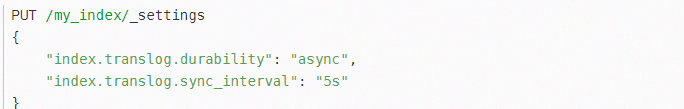
tanslog的作用就是给所有还没有flush到硬盘上的数据提供持久化记录，当es重启时，它首先会根据上一次停止时的commit point文件把所有已知的segments文件给恢复出来，然后再通过translog文件把上一次commit point之后的所有索引变化包括添加，删除，更新等操作给重放出来。

除此之外tanslog文件还用于提供一个近实时的CURD操作，当我们通过id读取，更新 除document时，es在从相关的segments里面查询d的优先访问最新版本的数据。

我们知道执行flush命令之后，所有系统cache中的数据会被同步到磁盘上并且会删除旧的translog然后生成新的translog，默认情况下es的shard会每隔30分钟自动执行一次flush命令，或者当translog变大超过一定的阈值后。

flush命令基本不需要我们手动操作，但当我们要重启节点或者关闭索引时，最好提前执行以下flush命令作为优化，因为es恢复索引或者重新打开索引时，它必须要先把translog里面的所有操作给恢复，所以也就是说translog越小，recovery恢复操作就越快。

如果在一个大数据量的集群中数据并不是很重要，那么就可以设置成每隔5秒进行异步fsync操作translog，配置如下：



## es的集群搭建

### 2.1 集群特性

#### 1、集群的节点类型

在Elasticsearch主要分成两类节点，一类是Master，一类是DataNode。

**Master节点：**

在Elasticsearch启动时，会选举出来一个Master节点。当某个节点启动后，然后

使用Zen Discovery机制找到集群中的其他节点，并建立连接。

discovery.seed\_hosts: [" 192.168.21.130 ", " 192.168.21.131 ", " 192.168.21.132 "]

并从候选主节点中选举出一个主节点。

cluster.initial\_master\_nodes: ["node1", "node2","node3"]

Master节点主要负责：

管理索引（创建索引、删除索引）、分配分片

维护元数据

管理集群节点状态

不负责数据写入和查询，比较轻量级

一个Elasticsearch集群中，只有一个Master节点。在生产环境中，内存可以相对

小一点，但机器要稳定。

**DataNode节点：**

在Elasticsearch集群中，会有N个DataNode节点。DataNode节点主要负责：

数据写入、数据检索，大部分Elasticsearch的压力都在DataNode节点上

在生产环境中，内存最好配置大一些

**集群状态**

**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

集群什么情况会处于一个yellow状态？

假设现在就一台linux服务器，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1个primary shard和1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。

测试：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。

**不同节点介绍**

主节点：node.master：true

数据节点: node.data: true

**1、客户端节点**

　　当主节点和数据节点配置都设置为false的时候，该节点只能处理路由请求，处理

搜索，分发索引操作等，从本质上来说该客户节点表现为智能负载平衡器。

独立的客户端节点在一个比较大的集群中是非常有用的，他协调主节点和数据节

点，客户端节点加入集群可以得到集群的状态，根据集群的状态可以直接路由请求。

**2、数据节点**

　　数据节点主要是存储索引数据的节点，主要对文档进行增删改查操作，聚合操作

等。数据节点对cpu，内存，io要求较高， 在优化的时候需要监控数据节点的状态，

当资源不够的时候，需要在集群中添加新的节点。

**3、主节点**

　主节点的主要职责是和集群操作相关的内容，如创建或删除索引，跟踪哪些节

点是群集的一部分，并决定哪些分片分配给相关的节点。稳定的主节点对集群的健康

是非常重要的，默认情况下任何一个集群中的节点都有可能被选为主节点，索引数据

和搜索查询等操作会占用大量的cpu，内存，io资源，为了确保一个集群的稳定，分离

主节点和数据节点是一个比较好的选择。

在一个生产集群中我们可以对这些节点的职责进行划分，建议集群中设置3台以上的节

点作为master节点，这些节点只负责成为主节点，维护整个集群的状态。再根据数据

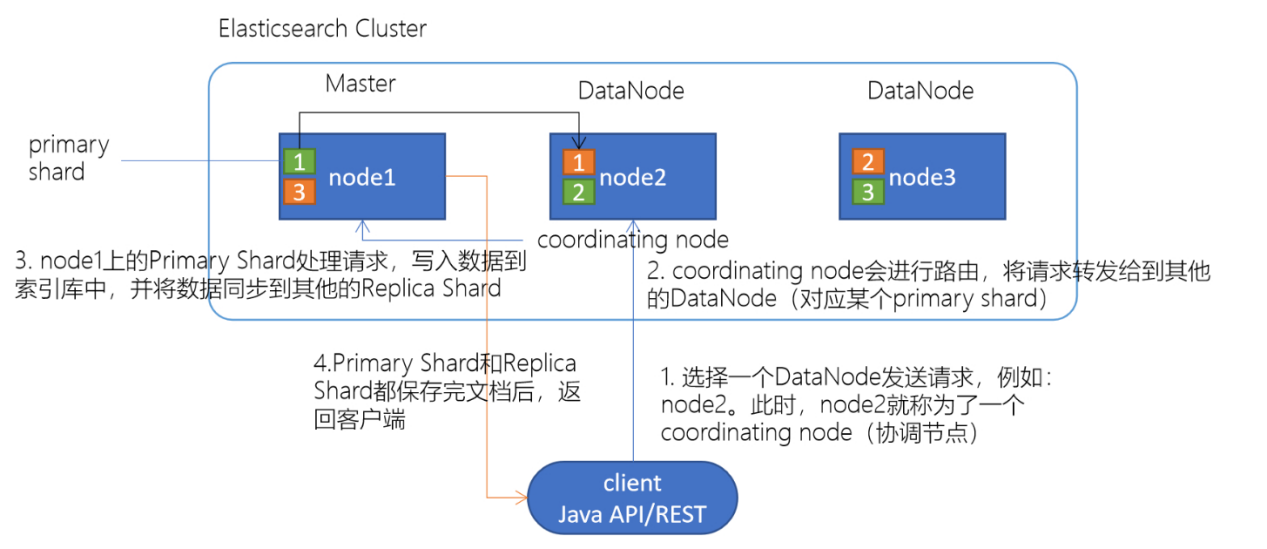
量设置一批data节点，这些节点只负责存储数据，后期提供建立索引和查询索引的服

务，这样的话如果用户请求比较频繁，这些节点的压力也会比较大，所以在集群中建

议再设置一批client节点(node.master: false node.data: false)，这些节点只负责

处理用户请求，实现请求转发，负载均衡等功能

#### 集群数据写入



1.选择任意一个DataNode发送请求，例如：node2。此时，node2就成为一个

coordinating node（协调节点）

2.计算得到文档要写入的分片`shard = hash(routing) % number\_of\_primary\_shards`

routing 是一个可变值，默认是文档的 \_id

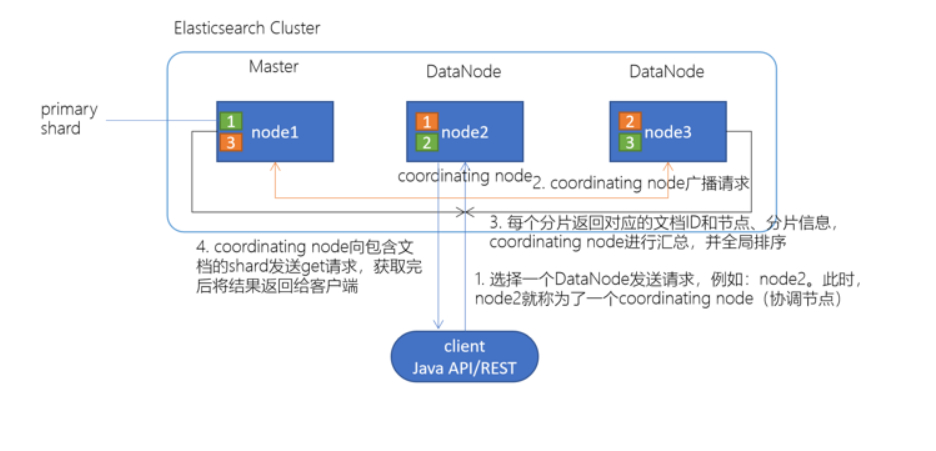
3.coordinating node会进行路由，将请求转发给对应的primary shard所在的DataNode（假设primary shard在node1、replica shard在node2）

4.node1节点上的Primary Shard处理请求，写入数据到索引库中，并将数据同步到

Replica shard

1. Primary Shard和Replica Shard都保存好了文档，返回client

#### 3、集群搜索



client发起查询请求，某个DataNode接收到请求，该DataNode就会成为协调节点

（Coordinating Node）

协调节点（Coordinating Node）将查询请求广播到每一个数据节点，这些数据节

点的分片会处理该查询请求

每个分片进行数据查询，将符合条件的数据放在一个优先队列中，并将这些数据

的文档ID、节点信息、分片信息返回给协调节点

协调节点将所有的结果进行汇总，并进行全局排序

协调节点向包含这些文档ID的分片发送get请求，对应的分片将文档数据返回给协

调节点，最后协调节点将数据返回给客户端

### 2.2 集群节点的关闭与重启



节点关闭或者索引关闭前执行以下flush操作，避免translog恢复占用太多时间，重启前要关闭集群的reblance（生产中建议关闭reblance功能，避免因为网络抖动等原因造成数据的无用复制），否则重启节点后failed掉的节点会立即开始在其他节点之间开始复制，对恢复速度有影响，用 ***GET  /\_cat/recovery?active\_only=true***确认下高耗时的recovery是哪种类型，是否的确在拷贝文件，还是做translog recovery。

用GET /\_cat/pending\_tasks 确认下master是否有大量的pending tasks在排队等待处理，如果pending tasks很多，更改集群设置的操作有可能会超时没生效，所以GET一下集群的settings看更改是否生效。

集群节点不要太多，避免小的分片的存在，否则新结点加入时master需要重新对unassinged的shard做allocation路由计算，这个过程需要遍历所有unassigned的shard，并发请求到所有结点，询问是否有关于此shard的信息，还需要更新集群的状态数据，并且下发给所有结点。 当shard太多的时候，这个过程会很慢，并且可能会阻塞读取结点状态信息的请求，以至于xpack都无法获取到监控数据。master在做集群状态数据更新的时候，是单线程的，如果有很多状态更新，会非常非常慢。

Recovery是指将一个索引的未分配shard分配到一个结点的过程。 在快照恢复，更改索引复制片数量，结点故障或者结点启动时发生。由于master持有整个集群的状态信息，因此可以判断出哪些shard需要做再分配，以及分配到哪个结点。例如:

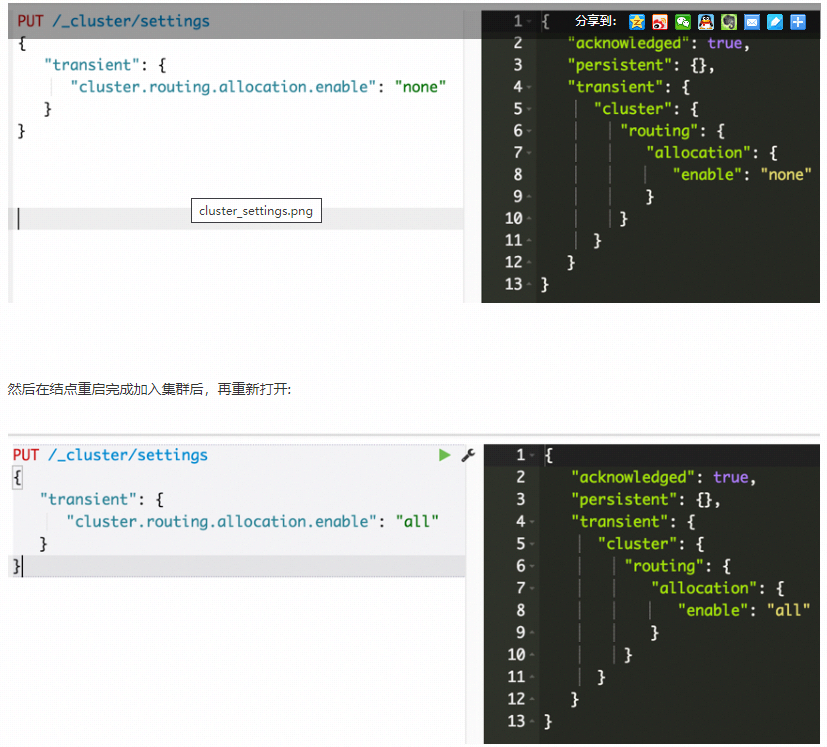
* 如果某个shard主片在，副片所在结点挂了，那么选择另外一个可用结点，将副片分配(allocate)上去，然后进行主从片的复制。
* 如果某个shard的主片所在结点挂了，副片还在，那么将副片升级为主片，然后做主副复制。
* 如果某个shard的主副片所在结点都挂了，则暂时无法恢复，等待持有相关数据的结点重新加入集群后，从结点上恢复主分片，再选择某个结点分配复制片，并从主分片同步数据。
* 

通过此命令查看集群状态

Recovery过程要消耗额外的资源，CPU、内存、结点之间的网络带宽等等。 这些额外的资源消耗，有可能会导致集群的服务能力降级，或者一部分功能暂时不可用。了解一些Recovery的过程和相关的配置参数，对于减小recovery带来的资源消耗，加快集群恢复过程都是很有帮助的。  
  
**减少集群Full Restart造成的数据来回拷贝**  
集群可能会有整体重启的需要，比如需要升级硬件、升级操作系统或者升级ES大版本。重启所有结点可能带来的一个问题: 某些结点可能先于其他结点加入集群。先加入集群的结点可能已经可以选举好master，并立即启动了recovery的过程，由于这个时候整个集群数据还不完整，master会指示一些结点之间相互开始复制数据。 那些晚到的结点，一旦发现本地的数据已经被复制到其他结点，则直接删除掉本地“失效”的数据。当整个集群恢复完毕后，数据分布不均衡显然是不均衡的，master会触发rebalance过程，将数据在结点之间挪动。整个过程无谓消耗了大量的网络流量。合理设置recovery相关参数则可以防范这种问题的发生。  
  
gateway.expected\_nodes  
gateway.expected\_master\_nodes  
gateway.expected\_data\_nodes

说集群里一旦有多少个结点就立即开始recovery过程。不同之处在于，第一个参数指的是master或者data都算在内，而后面两个参数则分指master和data node。  
  
在期待的节点数条件满足之前, recovery过程会等待gateway.recover\_after\_time (默认5分钟) 这么长时间，一旦等待超时，则会根据以下条件判断是否启动:  
gateway.recover\_after\_nodes  
gateway.recover\_after\_master\_nodes  
gateway.recover\_after\_data\_nodes  
  
举例来说，对于一个有10个data node的集群，如果有以下的设置:  
gateway.expected\_data\_nodes: 10  
gateway.recover\_after\_time: 5m  
gateway.recover\_after\_data\_nodes: 8

那么集群5分钟以内10个data node都加入了，或者5分钟以后8个以上的data node加入了，都会立即启动recovery过程。  
  
**减少主副本之间的数据复制**  
如果不是full restart，而是重启单个data node，仍然会造成数据在不同结点之间来回复制。为避免这个问题，可以在重启之前，先关闭集群的shard allocation与reblance（新增节点不要关闭）:



这样在结点重启完成后，尽量多的从本地直接恢复数据。  
但是在ES1.6版本之前，即使做了以上措施，仍然会发现有大量主副本之间的数据拷贝。从表面去看，这点很让人不能理解。 主副本数据完全一致，ES应该直接从副本本地恢复数据就好了，为什么要重新从主片再复制一遍呢？ 原因在于Recovery是简单对比主副本的segment file来判断哪些数据一致可以本地恢复，哪些不一致需要远端拷贝的。而不同结点的segment merge是完全独立运行的，可能导致主副本merge的深度不完全一样，从而造成即使文档集完全一样，产生的segment file却不完全一样。  
为了解决这个问题，ES1.6版本以后加入了synced flush的新特性。 对于5分钟没有更新过的shard，会自动synced flush一下，实质是为对应的shard加了一个synced flush ID。这样当重启结点的时候，先对比一下shard的synced flush ID，就可以知道两个shard是否完全相同，避免了不必要的segment file拷贝，极大加快了冷索引的恢复速度。  
需要注意的是synced flush只对冷索引有效，对于热索引（5分钟内有更新的索引）没有作用。 如果重启的结点包含有热索引，那么还是免不了大量的文件拷贝。因此在重启一个结点之前，最好按照以下步骤执行，recovery几乎可以瞬间完成:

1. 暂停数据写入程序
2. 关闭集群shard allocation
3. 手动执行POST /\_flush/synced
4. 重启结点
5. 重新开启集群shard allocation
6. 等待recovery完成，集群health status变成green
7. 重新开启数据写入程序

**(特别大的)热索引为何恢复慢**  
对于冷索引，由于数据不再更新，利用synced flush特性，可以快速直接从本地恢复数据。 而对于热索引，特别是shard很大的热索引，除了synced flush派不上用场需要大量跨结点拷贝segment file以外，translog recovery是导致慢的更重要的原因。  
  
从主片恢复数据到副片需要经历3个阶段:

1、对主片上的segment file做一个快照，然后拷贝到复制片分配到的结点。数据拷贝期间，不会阻塞索引请求，新增索引操作记录到translog里。

2、对translog做一个快照，此快照包含第一阶段新增的索引请求，然后重放快照里的索引操作。此阶段仍然不阻塞索引请求，新增索引操作记录到translog里。

3、为了能达到主副片完全同步，阻塞掉新索引请求，然后重放阶段二新增的translog操作。  
可见，在recovery完成之前，translog是不能够被清除掉的（禁用掉正常运作期间后台的flush操作）。如果shard比较大，第一阶段耗时很长，会导致此阶段产生的translog很大。重放translog比起简单的文件拷贝耗时要长得多，因此第二阶段的translog耗时也会显著增加。等到第三阶段，需要重放的translog可能会比第二阶段还要多。 而第三阶段是会阻塞新索引写入的，在对写入实时性要求很高的场合，就会非常影响用户体验。 因此，要加快大的热索引恢复速度，最好的方式是遵从上一节提到的方法: 暂停新数据写入，手动sync flush，等待数据恢复完成后，重新开启数据写入，这样可以将数据延迟影响可以降到最低。  
  
万一遇到Recovery慢，想知道进度怎么办呢？ CAT Recovery API可以显示详细的recovery各个阶段的状态。 这个API怎么用就不在这里赘述了，参考: <https://www.elastic.co/guide/en/elasticsearch/reference/current/cat-recovery.html>  
**其他Recovery相关的专家级设置**  
还有其他一些专家级的设置（参见： <https://www.elastic.co/guide/en/elasticsearch/reference/current/recovery.html>）可以影响recovery的速度，但提升速度的代价是更多的资源消耗，因此在生产集群上调整这些参数需要结合实际情况谨慎调整，一旦影响应用要立即调整回来。 对于搜索并发量要求高，延迟要求低的场合，默认设置一般就不要去动了。 对于日志实时分析类对于搜索延迟要求不高，但对于数据写入延迟期望比较低的场合，可以适当调大indices.recovery.max\_bytes\_per\_sec，提升recovery速度，减少数据写入被阻塞的时长。  
   
最后要说的一点是ES的版本迭代很快，对于Recovery的机制也在不断的优化中。 其中有一些版本甚至引入了一些bug，比如在ES1.4.x有严重的translog recovery bug，导致大的索引trans log recovery几乎无法完成 （[issue #9226](https://github.com/elastic/elasticsearch/issues/9226" \t "_blank)）  。因此实际使用中如果遇到问题，最好在Github的issue list里搜索一下，看是否使用的版本有其他人反映同样的问题。

### 2.3 集群脑裂

所谓脑裂问题，就是同一个集群中的不同节点，对于集群的状态有了不一样的理解，比如集群中存在两个master，如果因为网络的故障，导致一个集群被划分成了两片，每片都有多个node，以及一个master，那么集群中就出现了两个master了。

但是因为master是集群中非常重要的一个角色，主宰了集群状态的维护，以及shard的

分配，因此如果有两个master，可能会导致破坏数据。

如：

节点1在启动时被选举为主节点并保存主分片标记为0P，而节点2保存复制分片标记为

0R，现在，如果在两个节点之间的通讯中断了，会发生什么？由于网络问题或只是因为其

中一个节点无响应,这是有可能发生的。

两个节点都相信对方已经挂了。节点1不需要做什么，因为它本来就被选举为主节点。但是

节点2会自动选举它自己为主节点，因为它相信集群的一部分没有主节点了。

在elasticsearch集群，是有主节点来决定将分片平均的分布到节点上的。节点2保存的是

复制分片，但它相信主节点不可用了。所以它会自动提升复制节点为主节点。

现在我们的集群在一个不一致的状态了。打在节点1上的索引请求会将索引数据分配在主节

点，同时打在节点2的请求会将索引数据放在分片上。在这种情况下，分片的两份数据分开

了，如果不做一个全量的重索引很难对它们进行重排序。在更坏的情况下，一个对集群无

感知的索引客户端（例如，使用REST接口的）,这个问题非常透明难以发现，无论哪个节点

被命中索引请求仍然在每次都会成功完成。问题只有在搜索数据时才会被隐约发现：取决

于搜索请求命中了哪个节点，结果都会不同。

那么quorum参数的作用，就是告诉es直到有足够的master候选节点时，才可以选举出一个master，否则就不要选举出一个master。这个参数必须被设置为集群中master候选节点的quorum数量，也就是大多数。至于quorum的算法，就是：master候选节点数量 / 2 + 1。

比如我们有10个节点，都能维护数据，也可以是master候选节点，那么quorum就是10 / 2+ 1 = 6。

如果我们有三个master候选节点，还有100个数据节点，那么quorum就是3 / 2 + 1 = 2

如果我们有2个节点，都可以是master候选节点，那么quorum是2 / 2 + 1 = 2。此时就有问题了，因为如果一个node挂掉了，那么剩下一个master候选节点，是无法满足quorum数量的，也就无法选举出新的master，集群就彻底挂掉了。此时就只能将这个参数设置为1，

但是这就无法阻止脑裂的发生了。

2个节点，discovery.zen.minimum\_master\_nodes分别设置成2和1会怎么样综上所述，一个生产环境的es集群，至少要有3个节点，同时将这个参数设置为quorum，也就是2。discovery.zen.minimum\_master\_nodes设置为2，如何避免脑裂呢？

那么这个是参数是如何避免脑裂问题的产生的呢？比如我们有3个节点，quorum是2.现在网络故障，1个节点在一个网络区域，另外2个节点在另外一个网络区域，不同的网络区域内无法通信。这个时候有两种情况情况：

（1）如果master是单独的那个节点，另外2个节点是master候选节点，那么此时那个单独的master节点因为没有指定数量的候选master node在自己当前所在的集群内，因此就会取消当前master的角色，尝试重新选举，但是无法选举成功。然后另外一个网络区域内的node因为无法连接到master，就会发起重新选举，因为有两个master候选节点，满足了quorum，因此可以成功选举出一个master。此时集群中就会还是只有一个master。

（2）如果master和另外一个node在一个网络区域内，然后一个node单独在一个网络区域

内。那么此时那个单独的node因为连接不上master，会尝试发起选举，但是因为master候选节点数量不到quorum，因此无法选举出master。而另外一个网络区域内，原先的那个

master还会继续工作。这也可以保证集群内只有一个master节点。

综上所述，集群中master节点的数量至少3台，三台主节点通过在elasticsearch.yml中配置discovery.zen.minimum\_master\_nodes: 2，就可以避免脑裂问题的产生。

## 1.3索引的创建与数据添加

**参考资料：**

<https://www.elastic.co/guide/cn/elasticsearch/guide/cn/index.html>

这个是中文版的，虽然这个是es2.x版本的，但是大家可以发现其核心的技术点并没有太多的变化，仍然具备参考价值。

<https://www.elastic.co/guide/en/elasticsearch/reference/6.0/search-aggregations-metrics-avg-aggregation.html>

6.x英文版api资料

Es名词定义：

对外提供检索服务，http或者transport协议对外提供搜索。Restful的json。Es6.6.0 es2.x。

索引=数据库：很多个

类型（type）=表 =》 es6.x只有一个type，之前可以建很多，es7.x就没有这个type了（\_doc）。

文档=行数据 docment。

Filed

缺点：nosql 非关系型的，没有办法链接查询的，也就是跨索引查询。



Es基础类型：

Text：字符串类型，在写入时进行分词，然后插入倒排索引；

Keyword：不能被分词，只可以精确匹配的字符串类型，在写入时在整个词插入倒排索引

Date：日期类型，通常配合format使用 比如{“type”:”date”,”format”:”yyyy-MM-dd”}

Long,integer,short…

Boolean

Array：数组类型

Object：一般是json

Ip：ip地址

geo\_point：地理位置 {

“lat”:

“lon”:

}

Put重建索引只会更新数据，不会改变分片副本等配置值

指定分片与副本

PUT /test2

{

"settings": {

"number\_of\_shards": 1,

"number\_of\_replicas": 1

}

}

更新副本数量

PUT /test2/\_settings

{

"number\_of\_replicas": 0

}

添加数据

PUT /test/\_doc/1

{

"name":"kevin",

"age1":"30"

}

结构化索引创建

PUT /test

{

"settings": {

"number\_of\_shards": 1,

"number\_of\_replicas": 1

},

"mappings": {

"\_doc":{

"properties":{

"name":{

"type":"text","analyzer":"ik\_max\_word","search\_analyzer":"ik\_smart"

},

"sname":{

"type":"text","analyzer":"ik\_smart"

},

"enname":{

"type":"text","analyzer":"english"

},

"age":{

"type":"intrger"

}

}

}

}

}

同义词部分的索引

PUT /test11

{

"settings": {

"number\_of\_replicas": 1,

"number\_of\_shards": 1,

"analysis": {

"filter": {

"my\_synonym\_filter": {

"type": "synonym",

"synonyms\_path": "analysis-ik/synonyms.txt"

}

},

"analyzer": {

"ik\_syno": {

"type": "custom",

"tokenizer": "ik\_smart",

"filter": [

"my\_synonym\_filter"

]

},

"ik\_syno\_max": {

"type": "custom",

"tokenizer": "ik\_max\_word",

"filter": [

"my\_synonym\_filter"

]

}

}

}

},

"mappings": {

"\_doc": {

"properties": {

"name": {

"type": "text",

"analyzer": "ik\_syno\_max",

"search\_analyzer": "ik\_syno"

}

}

}

}

}

使用put会重建索引，如果只修改数据可以使用post

POST /test/\_doc/1/\_update

{

"doc":{

"name":"kevin1"

}

}

如果post不加\_update会更新所有数据等同于put

POST /test/\_doc/2

{

"age":3

}

加上\_create后只能执行一次，否则会报错

POST /test/\_doc/3/\_create

{

"age":3

}

当post创建数据不指定id时，es会指定一个默认ID，并且可以重复创建

POST /test/\_doc{

"age":3

}

{

"\_index" : "test",

"\_type" : "\_doc",

"\_id" : "PmzHeW8BTfPcbtGGONN4",

"\_score" : 1.0,

"\_source" : {

"age" : 4

}

# 数据操作

## 2.1 restFul 操作

Restful是一种面向资源的架构风格，可以简单理解为：使用URL定位资源，用HTTP动词（GET,POST,DELETE,PUT）描述操作。 基于Restful API ES和所有客户端的交互都是使用JSON格式的数据.其他所有程序语言都可以使用RESTful API，通过9200端口的与ES进行通信

GET查询

PUT添加

POST修改

DELE删除

使用Restful的好处：

透明性，暴露资源存在。

充分利用 HTTP 协议本身语义，不同请求方式进行不同的操作

### 1、查询

**1、查询当前类型中的所有文档 \_search**

格式: GET /索引名称/类型/\_search

举例: GET /es\_db/\_doc/\_search

SQL:  select \* from student

**2、条件查询, 如要查询age等于28岁的 \_search?q=\*:\*\*\***

格式: GET /索引名称/类型/\_search?q=\*:\*\*\*

举例: GET /es\_db/\_doc/\_search?q=age:28

SQL:  select \* from student where age = 28

**3、范围查询, 如要查询age在25至26岁之间的 \_search?q=\*\*\*[\*\* TO \*\*]  注**

**意: TO 必须为大写**

格式: GET /索引名称/类型/\_search?q=\*\*\*[25 TO 26]

举例: GET /es\_db/\_doc/\_search?q=age[25 TO 26]

SQL:  select \* from student where age between 25 and 26

**4、根据多个ID进行批量查询 \_mget**

格式: GET /索引名称/类型/\_mget

举例: GET /es\_db/\_doc/\_mget

{

"ids":["1","2"]

 }

SQL:  select \* from student where id in (1,2)

**5、查询年龄小于等于28岁的 :<=**

格式: GET /索引名称/类型/\_search?q=age:<=\*\*

举例: GET /es\_db/\_doc/\_search?q=age:<=28

SQL:  select \* from student where age <= 28

**6、分页查询 from=\*&size=\***

格式: GET /索引名称/类型/\_search?q=age[25 TO 26]&from=0&size=1

举例: GET /es\_db/\_doc/\_search?q=age[25 TO 26]&from=0&size=1

SQL:  select \* from student where age between 25 and 26 limit 0, 1

**7、对查询结果只输出某些字段 \_source=字段,字段**

格式: GET /索引名称/类型/\_search?\_source=字段,字段

举例: GET /es\_db/\_doc/\_search?\_source=name,age

SQL:  select name,age from student

**8、对查询结果排序 sort=字段:desc/asc**

格式: GET /索引名称/类型/\_search?sort=字段 desc

举例: GET /es\_db/\_doc/\_search?sort=age:desc

SQL:  select \* from student order by age desc

### 2、批量操作

1、批量获取文档数据

三种情况：URL中不指定index与type、指定index不指定type、指定index与type

A、不指定index与type

GET \_mget

{

"docs": [

{

"\_index": "es\_db",

"\_type": "\_doc",

"\_id": 1

},

{

"\_index": "es\_db",

"\_type": "\_doc",

"\_id": 2

}

]

}

B、指定index不指定type

GET /user/\_mget

{

"docs": [

{

"\_type":"\_doc",

"\_id": 3

},

{

"\_type":"\_doc",

"\_id": 4

}

]

}

C、指定index与type

GET /es\_db/\_doc/\_mget

{

"docs": [

{

"\_id": 1

},

{

"\_id": 2

}

]

}

2、批量增删改

批量对文档进行写操作是通过\_bulk的API来实现的

请求方式：POST

请求地址：\_bulk

请求参数：通过\_bulk操作文档，一般至少有两行参数(或偶数行参数)

第一行参数为指定操作的类型及操作的对象

(index,type和id)

第二行参数才是操作的数据

**参数结构：**

{"actionName":{"\_index":"indexName", "\_type":"typeName","\_id":"id"}}

{"field1":"value1", "field2":"value2"}

actionName：表示操作类型，主要有create,index,delete和update

(1)批量创建文档create

POST \_bulk

{"create":{"\_index":"article", "\_type":"\_doc", "\_id":3}}

{"id":3,"title":"白起老师1","content":"白起老师666","tags":["java", "面向对

象"],"create\_time":1554015482530}

{"create":{"\_index":"article", "\_type":"\_doc", "\_id":4}}

{"id":4,"title":"白起老师2","content":"白起老师NB","tags":["java", "面向对

象"],"create\_time":1554015482530}

(2)普通创建或全量替换index

POST \_bulk

{"index":{"\_index":"article", "\_type":"\_doc", "\_id":3}}

{"id":3,"title":"图灵徐庶老师(一)","content":"图灵学院徐庶老师666","tags":["j

ava", "面向对象"],"create\_time":1554015482530}

{"index":{"\_index":"article", "\_type":"\_doc", "\_id":4}}

{"id":4,"title":"图灵诸葛老师(二)","content":"图灵学院诸葛老师NB","tags":["ja

va", "面向对象"],"create\_time":1554015482530}

如果原文档不存在，则是创建

如果原文档存在，则是替换(全量修改原文档)

(3)批量删除delete

POST \_bulk

{"delete":{"\_index":"article", "\_type":"\_doc", "\_id":3}}

{"delete":{"\_index":"article", "\_type":"\_doc", "\_id":4}}

(4)批量修改update

POST \_bulk

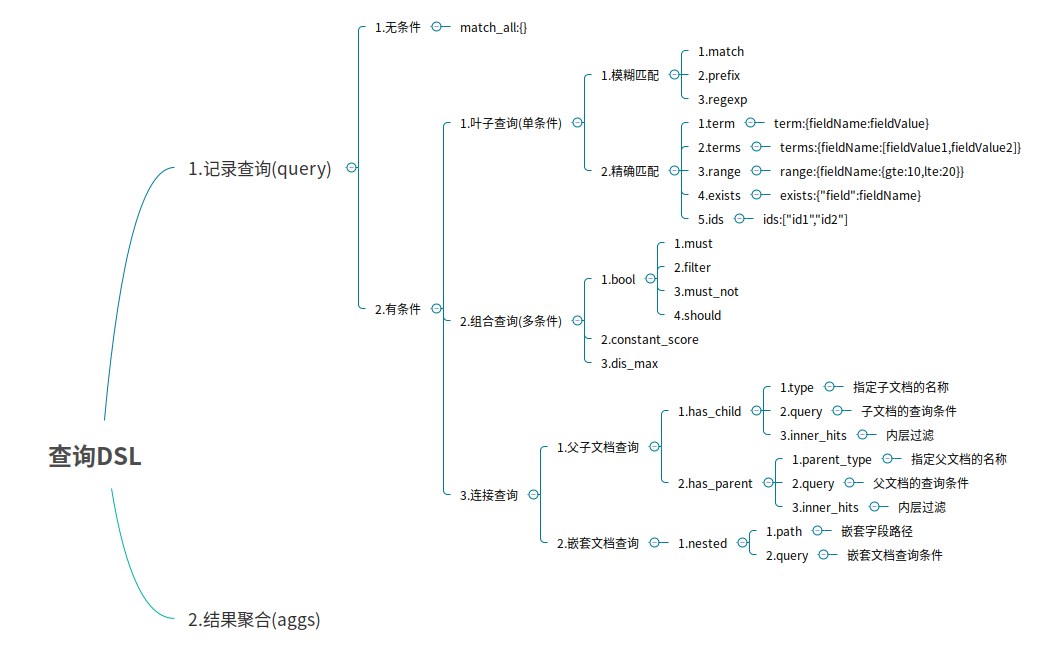
{"update":{"\_index":"article", "\_type":"\_doc", "\_id":3}}

{"doc":{"title":"ES大法必修内功"}}

{"update":{"\_index":"article", "\_type":"\_doc", "\_id":4}}

{"doc":{"create\_time":1554018421008}}

## 2.2 kibana查询操作



Es的DSL查询可以分为两种，查询DSL与过滤DSL



**query DSL**

在查询上下文中，查询会回答这个问题——“这个文档匹不匹配这个查询，它的

相关度高么？”

如何验证匹配很好理解，如何计算相关度呢？ES中索引的数据都会存储一个

\_score分值，分值越高就代表越匹配。另外关于某个搜索的分值计算还是很复杂

的，因此也需要一定的时间。

**filter DSL**

在过滤器上下文中，查询会回答这个问题——“这个文档匹不匹配？”

答案很简单，是或者不是。它不会去计算任何分值，也不会关心返回的排序问

题，因此效率会高一点。

过滤上下文 是在使用filter参数时候的执行环境，比如在bool查询中使用

must\_not或者filter

另外，经常使用过滤器，ES会自动的缓存过滤器的内容，这对于查询来说，会提

高很多性能。

### 记录查询

无查询条件是查询所有，默认是查询所有的，或者使用match\_all表示所

有

GET /user/\_doc/\_search

{

"query":{

"match\_all":{}

}

}

有条件查询分为三种：叶子查询（单条件）、组合查询（多条件）、连接查询（父子文档查询、嵌套文档查询）

#### 1、叶子查询

叶子查询可分为模糊查询、精确匹配两种

总结:

1. match

match：模糊匹配，需要指定字段名，但是输入会进行分词，比如"helloworld"会进行拆分为hello和world，然后匹配，如果字段中包含hello或者world，或者都包含的结果都会被查询出来，也就是说match是一个部分匹配的模糊查询。查询条件相对来说比较宽松。

2. term

term: 这种查询和match在有些时候是等价的，比如我们查询单个的词hello，那么会和match查询结果一样，但是如果查询"hello world"，结果就相差很大，因为这个输入不会进行分词，就是说查询的时候，是查询字段分词结果中是否有"hello world"的字样，而不是查询字段中包含"hello world"的字样。当保存数据"hello world"时，elasticsearch会对字段内容进行分词，"hello world"会被分成hello和world，不存在"hello world"，因此这里的查询结果会为空。这也是term查询和match的区别。

3. match\_phase

match\_phase：会对输入做分词，但是需要结果中也包含所有的分词，而且顺序要求一样。以"hello world"为例，要求结果中必须包含hello和world，而且还要求他们是连着的，顺序也是固定的，hello that word不满足，world hello也不满足条件。

4. query\_string

query\_string：和match类似，但是match需要指定字段名，query\_string是在所

有字段中搜索，范围更广泛。

term查询keyword字段：

 term不会分词。而keyword字段也不分词。需要完全匹配才可。

 hello world  ===  hello       world

 hello world

term查询text字段：

因为text字段会分词，而term不分词，所以term查询的条件必须是text字段分词后的某一个。

match查询keyword字段：

match会被分词，而keyword不会被分词，match的需要跟keyword的完全匹配可以。

match查询text字段：

match分词，text也分词，只要match的分词结果和text的分词结果有相同的就匹配。

##### 1、模糊匹配

模糊匹配主要是针对文本类型的字段，文本类型的字段会对内容进行分

词，对查询时，也会对搜索条件进行分词，然后通过倒排索引查找到匹

配的数据，模糊匹配主要通过match等参数来实现

match : 通过match关键词模糊匹配条件内容

prefix : 前缀匹配

regexp : 通过正则表达式来匹配数据

**Match的用法：**

match条件还支持以下参数：

query : 指定匹配的值

operator : 匹配条件类型

and : 条件分词后都要匹配

or : 条件分词后有一个匹配即可(默认)

minmum\_should\_match : 指定最小匹配的数量

POST /es\_db/\_doc/\_search{

"from": 0,

"size": 2,

"query": {

"match": {

"address": "广州"

}}}

功能相当于：select \* from user where address like '%广州%' limit 0, 2

###### 多字段模糊匹配查询multi\_match

POST /es\_db/\_doc/\_search{

"query":{

"multi\_match":{

"query":"张三",

"fields":["address","name"]

}}}

select \* from student  where name like '%张三%' or address like '%张三%'

**多字段查询策略：best fields、most fields、cross fields**

best fields策略： 搜索的document中的某一个field，尽可能多的匹配搜索条件。与之相反的是，尽可能多的字段匹配到搜索条件（most fields策略）。如百度搜索使用这种策略。

best fields取两边分词分数高的那个字段，多字段查询中如果字段中匹配到多个分词，则此字段分数为多个分词分数之和

**优点**：精确匹配的数据可以尽可能的排列在最前端，且可以通过minimum\_should\_match来去除长尾数据，避免长尾数据字段对排序结果的影响。

长尾数据比如说我们搜索4个关键词，但很多文档只匹配1个，也显示出来了，这些文

档其实不是我们想要的

**缺点**：相对排序不均匀。

dis\_max语法： 直接获取搜索的多条件中的，单条件query相关度分数最

高的数据，以这个数据做相关度排序。

下述的案例中，就是找name字段中rod匹配相关度分数或remark字段中java

developer匹配相关度分数，哪个高，就使用哪一个相关度分数进行结果排序。

GET /es\_db/\_search{

"query": {

"dis\_max": {

"queries": [{

"match": {

"name": "rod"

}},

{

"match": {

"remark": "java developer"

}}]}}}

基于tie\_breaker参数优化dis\_max搜索效果

dis\_max是将多个搜索query条件中相关度分数最高的用于结果排序，忽略其他

query分数，在某些情况下，可能还需要其他query条件中的相关度介入最终的结果

排序，这个时候可以使用tie\_breaker参数来优化dis\_max搜索。tie\_breaker参数

代表的含义是：将其他query搜索条件的相关度分数乘以参数值，再参与到结果排

序中。如果不定义此参数，相当于参数值为0。所以其他query条件的相关度分数被

忽略。

使用multi\_match语法为：其中type常用的有best\_fields和most\_fields。^n代表权重，

相当于"boost":n。

GET /es\_db/\_search{

"query": {

"multi\_match": {

"query": "rod java developer",

"fields": ["name", "remark^2"],

"type": "best\_fields",

"tie\_breaker": 0.5,

"minimum\_should\_match" : "50%"

}}}

cross fields ：一个唯一的标识，分部在多个fields中，使用这种唯一标识

索数据就称为cross fields搜索。如：人名可以分为姓和名，地址可以分为省、

市、区县、街道等。那么使用人名或地址来搜索document，就称为cross fields搜

索。

实现这种搜索，一般都是使用most fields搜索策略。因为这就不是一个field

的问题。

Cross fields搜索策略，是从多个字段中搜索条件数据。默认情况下，和most

fields搜索的逻辑是一致的，计算相关度分数是和best fields策略一致的。一般

来说，如果使用cross fields搜索策略，那么都会携带一个额外的参数operator。

用来标记搜索条件如何在多个字段中匹配。

以分词为单位，单个分词的得分取该分词在所有字段中的最大值，然后多个分词的得分之和为该条数据的总得分

当然，在ES中也有cross fields搜索策略。具体语法如下：

GET /es\_db/\_search{

"query": {

"multi\_match": {

"query": "java developer",

"fields": ["name", "remark"],

"type": "cross\_fields",

"operator" : "and"

}}}

上述语法代表的是，搜索条件中的java必须在name或remark字段中匹配，developer也必须在name或remark字段中匹配。

GET /es\_db/\_search{

"query": {

"multi\_match": {

"query": "java developer",

"fields": ["name", "remark"],

"type": "cross\_fields",

"operator" : "and"

}}}

most field策略问题：most fields策略是尽可能匹配更多的字段，所以会导致精确搜索结果排序问题。又因为cross fields搜索，不能使用minimum\_should\_match来去除长尾数据。

所以在使用most fields和cross fields策略搜索数据的时候，都有不同的缺陷。所以商业项目开发中，都推荐使用best fields策略实现搜索。

**operator :** 匹配条件类型，默认or，此时查询字段必须包含查询条件的两个分词才可以

GET /es\_db/\_search{

"query": {

"match": {

"remark": {

"query": "java developer",

"operator": "and"

}}}}

**minimum\_should\_match**：可以使用数字也可以使用百分比

可以使用百分比或固

定数字。百分比代表query搜索条件中词条百分比，如果无法整除，向下匹配

（如，query条件有3个单词，如果使用百分比提供精准度计算，那么是无法除尽

的，如果需要至少匹配两个单词，则需要用67%来进行描述。如果使用66%描述，ES

则认为匹配一个单词即可。）

GET /es\_db/\_search{

"query": {

"match": {

"remark": {

"query": "java architect assistant",

"minimum\_should\_match": "68%"

}}}}

使用bool 与should更细致的控制匹配条件

GET /es\_db/\_search{

"query": {

"bool": {

"should": [{

"match": {

"remark": "java"

}},

{

"match": {

"remark": "developer"

}},

{

"match": {

"remark": "assistant"

}}],

"minimum\_should\_match": 2

}}}

###### copy\_to组合fields

京东中，如果在搜索框中输入“手机”，点击搜索，那么是在商品的类型名称、商品的名称、商品的卖点、商品的描述等字段中，哪一个字段内进行数据的匹配？如果使用某一个字段做搜索不合适，那么使用\_all做搜索是否合适？也不合适，因为\_all字段中可能包含图片，价格等字段。假设，有一个字段，其中的内容包括(但不限于)：商品类型名称、商品名称、

商品卖点等字段的数据内容。是否可以在这个特殊的字段上进行数据搜索匹配？

{

"category\_name" : "手机",

"product\_name" : "一加6T手机",

"price" : 568800,

"sell\_point" : "国产最好的Android手机",

"tags": ["8G+128G", "256G可扩展"],

"color" : "红色",

"keyword" : "手机 一加6T手机 国产最好的Android手机"

}

copy\_to : 就是将多个字段，复制到一个字段中，实现一个多字段组合。copy\_to

可以解决cross fields搜索问题，在商业项目中，也用于解决搜索条件默认字段问

题。

如果需要使用copy\_to语法，则需要在定义index的时候，手工指定mapping映射策

略。

copy\_to语法：

PUT /es\_db/\_mapping{

"properties": {

"provice" : {

"type": "text",

"analyzer": "standard",

"copy\_to": "address"

},

"city" : {

"type": "text",

"analyzer": "standard",

"copy\_to": "address"

},

"street" : {

"type": "text",

"analyzer": "standard",

"copy\_to": "address"

},

"address" : {

"type": "text",

"analyzer": "standard"

}}}

上述的mapping定义中，是新增了4个字段，分别是provice、city、street、address，其中provice、city、street三个字段的值，会自动复制到address字段中，实现一个字段的组合。那么在搜索地址的时候，就可以在address字段中做条件匹配，从而避免most fields策略导致的问题。在维护数据的时候，不需对address字段特殊的维护。因为address字段是一个组合字段，是由ES自动维护的。类似java代码中的推导属性。在存储的时候，未必存在，但是在逻辑上是一定存在的，因为address是由3个物理存在的属性province、city、street组成的。

###### match phrase近似匹配

使用match phrase做搜

索的时候，也是和match类似，首先对搜索条件进行分词-analyze。将搜索条件拆

分成hello和world。既然是分词后再搜索，ES是如何实现短语搜索的？

这里涉及到了倒排索引的建立过程。在倒排索引建立的时候，ES会先对

document数据进行分词，如：

GET \_analyze{

"text": "hello world, java spark",

"analyzer": "standard"

}

分词的结果是：

{

"tokens": [

    {

     "token": "hello",

       "start\_offset": 0,

       "end\_offset": 5,

"type": "<ALPHANUM>",

      "position": 0

    },

    {

      "token": "world",

      "start\_offset": 6,

      "end\_offset": 11,

      "type": "<ALPHANUM>",

      "position": 1

    },

    {

     "token": "java",

      "start\_offset": 13,

      "end\_offset": 17,

      "type": "<ALPHANUM>",

      "position": 2

  },

    {

      "token": "spark",

      "start\_offset": 18,

      "end\_offset": 23,

      "type": "<ALPHANUM>",

      "position": 3

}]}

从上述结果中，可以看到。ES在做分词的时候，除了将数据切分外，还会保留一个position。position代表的是这个词在整个数据中的下标。当ES执行match phrase搜索的时候，首先将搜索条件hello world分词为hello和world。然后在倒排索引中检索数据，如果hello和world都在某个document的某个field出现时，那么检查这两个匹配到的单词的position是否是连续的，如果是连续的，代表匹配成功，如果是不连续的，则匹配失败。

**match phrase/freiz/搜索参数 -- slop**

在做搜索操作的是，如果搜索参数是hello spark。而ES中存储的数据是hello

world, java spark。那么使用match phrase则无法搜索到。在这个时候，可以使

用match来解决这个问题。但是，当我们需要在搜索的结果中，做一个特殊的要

求：hello和spark两个单词距离越近，document在结果集合中排序越靠前，这个时

候再使用match则未必能得到想要的结果。

ES的搜索中，对match phrase提供了参数slop。slop代表match phrase短语搜

索的时候，单词最多移动多少次，可以实现数据匹配。在所有匹配结果中，多个单

词距离越近，相关度评分越高，排序越靠前。

这种使用slop参数的match phrase搜索，就称为近似匹配（proximity search）

如：

数据为： hello world, java spark

搜索为： match phrase : hello spark。

slop为： 3（代表单词最多移动3次。）

执行短语搜索的时候，将条件hello spark分词为hello和spark两个单词。并且连续。

hello spark接下来，可以根据slop参数执行单词的移动。

下标 ： 0 1 2 3

doc ： hello world java spark

搜索 ： hello spark

移动1： hello spark

移动2： hello spark

匹配成功，不需要移动第三次即可匹配。

如果：

数据为： hello world, java spark

搜索为： match phrase : spark hello。

slop为： 5 （代表单词最多移动5次。）

执行短语搜索的时候，将条件hello spark分词为hello和spark两个单词。并且连续。

spark hello

接下来，可以根据slop参数执行单词的移动。

下标 ： 0 1 2 3

doc ： hello world java spark

搜索 ： spark hello

移动1：spark/hello

移动2：hello spark

移动3：hello spark

移动4：hello spark

匹配成功，不需要移动第五次即可匹配。

如果当slop移动次数使用完毕，还没有匹配成功，则无搜索结果。如果使用中文分词，则

移动次数更加复杂，因为中文词语有重叠情况，很难计算具体次数，需要多次尝试才行。

如果当slop移动次数使用完毕，还没有匹配成功，则无搜索结果。如果使用中文分词，则

移动次数更加复杂，因为中文词语有重叠情况，很难计算具体次数，需要多次尝试才行。

使用match和proximity search实现召回率和精准度平衡。

召回率：召回率就是搜索结果比率，如：索引A中有100个document，搜索时返

回多少个document，就是召回率（recall）。

精准度：就是搜索结果的准确率，如：搜索条件为hello java，在搜索结果中

尽可能让短语匹配和hello java离的近的结果排序靠前，就是精准度

（precision）。

如果在搜索的时候，只使用match phrase语法，会导致召回率底下，因为搜索

结果中必须包含短语（包括proximity search）。

如果在搜索的时候，只使用match语法，会导致精准度底下，因为搜索结果排

序是根据相关度分数算法计算得到。

那么如果需要在结果中兼顾召回率和精准度的时候，就需要将match和

proximity search混合使用，来得到搜索结果。

GET /test\_a/\_search{

"query": {

"bool": {

"must": [{

"match": {

"f": "java spark"

}}],

"should": [{

"match\_phrase": {

"f": {

"query": "java spark",

"slop" : 50

}}}]}}}

###### boost权重控制

搜索document中remark字段中包含java的数据，如果remark中包含developer

或architect，则包含architect的document优先显示。（就是将architect数据匹

配时的相关度分数增加）。

一般用于搜索时相关度排序使用。如：电商中的综合排序。将一个商品的销量，广告投放，评价值，库存，单价比较综合排序。在上述的排序元素中，广告投放权重最高，库存权重最低。

GET /es\_db/\_search{

"query": {

"bool": {

"must": [{

"match": {

"remark": "java"

}}],

"should": [{

"match": {

"remark": {

"query": "developer",

"boost" : 1

}}},

{

"match": {

"remark": {

"query": "architect",

"boost" : 3

}}}]}}}

**Match底层语句转换：**

在ES中，执行match搜索的时候，ES底层通常都会对搜索条件进行底层转换，来实现最终的搜索结果

**FOR EXAMPLE:**

GET /es\_db/\_search{

"query": {

"match": {

"remark": "java developer"

}}}

转换后是：

GET /es\_db/\_search{

"query": {

"bool": {

"should": [{

"term": {

"remark": "java"

}},

{

"term": {

"remark": {

"value": "developer"

}}}]}}}

**条件为and的查询**：

GET /es\_db/\_search{

"query": {

"match": {

"remark": {

"query": "java developer",

"operator": "and"

}}}}

转换后是：

GET /es\_db/\_search{

"query": {

"bool": {

"must": [{

"term": {

"remark": "java"

}},

{

"term": {

"remark": {

"value": "developer"

}}}]}}}

GET /es\_db/\_search{

"query": {

"match": {

"remark": {

"query": "java architect assistant",

"minimum\_should\_match": "68%"

}}}}

转换后为：

GET /es\_db/\_search{

"query": {

"bool": {

"should": [

{

"term": {

"remark": "java"

}},

{

"term": {

"remark": "architect"

}},

{

"term": {

"remark": "assistant"

}}],

"minimum\_should\_match": 2

}}}

###### 前缀搜索 prefix search

GET /test\_a/\_search{

"query": {

"prefix": {

"f.keyword": {

"value": "J"

}}}}

针对前缀搜索，是对keyword类型字段而言。而keyword类型字段数据大小写敏感。

前缀搜索效率比较低。前缀搜索不会计算相关度分数。前缀越短，效率越低。

如果使用前缀搜索，建议使用长前缀。因为前缀搜索需要扫描完整的索引内容，所

以前缀越长，相对效率越高。

###### 通配符搜索

ES中也有通配符。但是和java还有数据库不太一样。通配符可以在倒排索引中

使用，也可以在keyword类型字段中使用。

常用通配符：

? - 一个任意字符

\* - 0~n个任意字符

GET /test\_a/\_search

{

"query": {

"wildcard": {

"f.keyword": {

"value": "?e\*o\*"

}}}}

性能也很低，也是需要扫描完整的索引。不推荐使用。

###### 正则搜索

ES支持正则表达式。可以在倒排索引或keyword类型字段中使用。

常用符号：

[] - 范围，如： [0-9]是0~9的范围数字

. - 一个字符

+ - 前面的表达式可以出现多次。

GET /test\_a/\_search{

"query": {

"regexp" : {

"f.keyword" : "[A‐z].+"

}}}

性能也很低，需要扫描完整索引。

###### 搜索推荐match\_phrase\_prefix

搜索推荐： search as your type， 搜索提示。如：索引中有若干数据

以“hello”开头，那么在输入hello的时候，推荐相关信息。（类似百度输入框）

语法：

GET /test\_a/\_search{

"query": {

"match\_phrase\_prefix": {

"f": {

"query": "java s",

"slop" : 10,

"max\_expansions": 10

}}}}

其原理和match phrase类似，是先使用match匹配term数据（java），然后在指定的slop移动次数范围内，前缀匹配（s），max\_expansions是用于指定prefix最多匹配多少个term（单词），超过这个数量就不再匹配了。

这种语法的限制是，只有最后一个term会执行前缀搜索。执行性能很差，毕竟最后一个term是需要扫描所有符合slop要求的倒排索引的term。

因为效率较低，如果必须使用，则一定要使用参数max\_expansions。

###### fuzzy模糊搜索技术

搜索的时候，可能搜索条件文本输入错误，如：hello world -> hello

word。这种拼写错误还是很常见的。fuzzy技术就是用于解决错误拼写的（在英文

中很有效，在中文中几乎无效。）。其中fuzziness代表value的值word可以修改多

少个字母来进行拼写错误的纠正（修改字母的数量包含字母变更，增加或减少字

母。）。f代表要搜索的字段名称。

GET /test\_a/\_search{

"query": {

"fuzzy": {

"f" : {

"value" : "word",

"fuzziness": 2

}}}}

###### 搜索提示、自动补全

**Term Suggester**，基于编辑距离，对analyze过的单个term去提供建议，并不会考虑多个term/词组之间的关系。quert -> query

**Phrase Suggester**，在Term Suggester的基础上，通过ngram以词组为单位返回建议。noble prize -> nobel prize

**Completion Suggester**，FST数据结构，类似Trie树，不用打开倒排，快速返回，前缀匹配

**Context Suggester**，在Completion Suggester的基础上，用于filter和boost

suggest search（completion suggest）：就是建议搜索或称为搜索建议，前缀搜所，必须满足前缀与搜索条件相同，也可以叫做自动完成-auto completion。类似百度中的搜索联想提示功能。ES实现suggest的时候，性能非常高，其构建的不是倒排索引，也不是正排索引，就是纯的用于进行前缀搜索的一种特殊的数据结构（tire树），而且会全部放在内存中，所以suggest search进行的前缀搜索提示，性能是非常高。需要使用suggest的时候，必须在定义index时，为其mapping指定开启suggest。具体如下：

PUT /movie{

"mappings": {

"properties" : {

"title" : {

"type": "text",

"analyzer": "ik\_max\_word",

"fields": {

"suggest" : {

"type" : "completion",

"analyzer": "ik\_max\_word"

}}},

"content": {

"type": "text",

"analyzer": "ik\_max\_word"

}}}}

PUT /movie/\_doc/1{

"title": "西游记电影系列",

"content": "西游记之月光宝盒将与2021年进行......"

}

PUT /movie/\_doc/2{

"title": "西游记文学系列",

"content": "某知名网络小说作家已经完成了大话西游同名小说的出版"

}

PUT /movie/\_doc/3{

"title": "西游记之大话西游手游",

"content": "网易游戏近日出品了大话西游经典IP的手游，正在火爆内测中"

}

suggest 搜索：

GET /movie/\_search{

"suggest": {

"my‐suggest" : {

"prefix" : "西游记",

"completion" : {

"field" : "title.suggest"

}}}}

###### 分页查询

**使用from和size来进行分页**

在执行查询时，可以指定from（从第几条数据开始查起）和size（每页返回多少条）数据，就可以轻松完成分页。

from = (page – 1) \* size

POST /es\_db/\_doc/\_search{

"from": 0,

"size": 2,

"query": {

"match": {

"address": "广州天河"

}}}

**使用scroll方式进行分页**

前面使用from和size方式，查询在1W-5W条数据以内都是OK的，但如果数据比较多的时候，会出现性能问题。Elasticsearch做了一个限制，不允许查询的是10000条以后的数据。如果要查询1W条以后的数据，需要使用Elasticsearch中提供的scroll游标来查询。

在进行大量分页时，每次分页都需要将要查询的数据进行重新排序，这样非常浪费性能。

使用scroll是将要用的数据一次性排序好，然后分批取出。性能要比from + size好得多。

使用scroll查询后，排序后的数据会保持一定的时间，后续的分页查询都从该快照取数据

即可。

第一次使用scroll分页查询

此处，我们让排序的数据保持1分钟，所以设置scroll为1m

GET /es\_db/\_search?scroll=1m{

"query": {

"multi\_match":{

"query":"广州长沙张三",

"fields":["address","name"]

}},

"size":100

}

执行后，我们注意到，在响应结果中有一项：

"\_scroll\_id":"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAZEWY2VQZXBia1JTVkdhTWkwSl9GaUYtQQ=="后续，我们需要根据这个\_scroll\_id来进行查询

第二次直接使用scroll id进行查询

GET \_search/scroll?scroll=1m{

"scroll\_id":"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAZoWY2VQZXBia1JTVkdhTWkwSl9GaUYtQQ=="}

##### 2、精确匹配

term : 单个条件相等，精确匹配不会对查询条件进行分词

POST /es\_db/\_doc/\_search{

"query": {

"term": {

"name": "admin"

}}}

terms : 单个字段属于某个值数组内的值

range : 字段属于某个范围内的值

exists : 某个字段的值是否存在

ids : 通过ID批量查询

未指定字段条件查询 query\_string , 含 AND 与 OR 条件

POST /es\_db/\_doc/\_search{

"query":{

"query\_string":{

"query":"(广州) OR 长沙"

}}}

指定字段条件查询 query\_string , 含 AND 与 OR 条件

POST /es\_db/\_doc/\_search{

"query":{

"query\_string":{

"query":"admin OR 长沙",

"fields":["name","address"]

}}}

范围查询：

range：范围关键字

gte 大于等于

lte 小于等于

gt 大于

lt 小于

now 当前时间

POST /es\_db/\_doc/\_search{

"query" : {

"range" : {

"age" : {

"gte":25,

"lte":28

}}}}

分页、输出字段、排序综合查询

POST /es\_db/\_doc/\_search{

"query" : {

"range" : {

"age" : {

"gte":25,

"lte":28

}}},

"from": 0,

"size": 2,

"\_source": ["name", "age", "book"],

"sort": {"age":"desc"}

}

#### 2、组合条件查询

组合条件查询是将叶子条件查询语句进行组合而形成的一个完整的查询条件

bool : 各条件之间有and,or或not的关系

must : 各个条件都必须满足，即各条件是and的关系

should : 各个条件有一个满足即可，即各条件是or的关系

must\_not : 不满足所有条件，即各条件是not的关系

filter : 不计算相关度评分，它不计算\_score即相关度评分，效率更高

constant\_score : 不计算相关度评分

must/filter/shoud/must\_not 等的子条件是通

过 term/terms/range/ids/exists/match 等叶子条件为参数的

注：以上参数，当只有一个搜索条件时，must等对应的是一个对象，当

是多个条件时，对应的是一个数组

Filter过滤器方式查询，它的查询不会计算相关性分值，也不会对结果进

行排序, 因此效率会高一点，查询的结果可以被缓存。

POST /es\_db/\_doc/\_search{

"query" : {

"bool" : {

"filter" : {

"term":{

"age":25

}}}}}

#### 3、geo

ES支持地理位置的搜索和聚合分析，可实现在指定区域内搜索数据、搜索指定地点附近的数据、聚合分析指定地点附近的数据等操作。

ES中如果使用地理位置搜索的话，必须提供一个特殊的字段类型。GEO -geo\_point。地理位置的坐标点。

1、定义geo point mapping

如果需要使用地址坐标，则需要定义一个指定的mapping类型。具体如下：

使用什么数据可以确定，地球上的一个具体的点？经纬度。

PUT /hotel\_app{

"mappings": {

"properties": {

"pin": {

"type": "geo\_point"

},

"name" : {

"type" : "text",

"analyzer": "ik\_max\_word"

}}}}

2、录入数据

新增一个基于geo point类型的数据，可以使用多种方式。

多种类型描述geo\_point类型字段的时候，在搜索数据的时候，显示的格式

和录入的格式是统一的。不影响搜索。任何数据描述的geo\_point类型字段，都

适用地理位置搜索。

数据范围要求：纬度范围是-90~90之间，经度范围是-180~180之间。经纬度

数据都是浮点数或数字串（数字组成的字符串），最大精度：小数点后7位。（常

用小数点后6位即可。）

基于对象：latitude：纬度、longitude：经度。语义清晰，建议使用。

PUT /hotel\_app/\_doc/1{

"name": "七天连锁酒店",

"pin" : {

"lat" : 40.12,

"lon" : ‐71.34

}}

基于字符串：依次定义纬度、经度。不推荐使用

PUT /hotel\_app/\_doc/2{

"name": "维多利亚大酒店",

"pin" : "40.99, ‐70.81"

}

基于数组：依次定义经度、纬度。不推荐使用

PUT /hotel\_app/\_doc/3{

"name": " 红树林宾馆",

"pin" : [40, ‐73.81]

}

3、搜索指定区域范围内的数据

总结：

矩形范围搜索：传入的top\_left和bottom\_right坐标点是有固定要求的。地图中以北作为top，南作为bottom，西作为left，东作为right。也就是top\_left应该从西北向东南。Bottom\_right应该从东南向西北。Top\_left的纬度应该大于bottom\_right的纬度，top\_left的经度应该小于bottom\_right的经度。

多边形范围搜索：对传入的若干点的坐标顺序没有任何的要求。只要传入若干地理位置坐标点，即可形成多边形。

搜索矩形范围内的数据

GET /hotel\_app/\_doc/\_search{

"query": {

"bool": {

"must": [{

"match\_all": {}}],

"filter": {

"geo\_bounding\_box": {

"pin": {

"top\_left" : {

"lat" : 41.73,

"lon" : ‐74.1

},

"bottom\_right" : {

"lat" : 40.01,

"lon" : ‐70.12

}}}}}}}

GET /hotel\_app/\_doc/\_search{

"query": {

"constant\_score": {

"filter": {

"geo\_bounding\_box": {

"pin": {

"top\_left": {

"lat": ‐70,

"lon": 39

},

"bottom\_right": {

"lat": ‐75,

"lon": 41

}}}}}}}

搜索多边形范围内的数据

1 GET /hotel\_app/\_doc/\_search

2 {

3 "query": {

4 "bool": {

5 "must": [

6 {

7 "match\_all": {}

8 }

9 ],

10 "filter": {

11 "geo\_polygon": {

12 "pin": {

13 "points": [

14 {"lat" : 40.73, "lon" : ‐74.1},

15 {"lat" : 40.01, "lon" : ‐71.12},

16 {"lat" : 50.56, "lon" : ‐90.58}

17 ]

18 }

19 }

20 }

21 }

22 }

23 }

4、搜索某地点附近的数据

这个搜索在项目中更加常用。类似附近搜索功能。

Distance距离的单位，常用的有米（m）和千米（km）。

建议使用filter来过滤geo\_point数据。因为geo\_point数据相关度评分计算

比较耗时。使用query来搜索geo\_point数据效率相对会慢一些。建议使用

filter。

1 GET /hotel\_app/\_doc/\_search

2 {

3 "query": {

4 "bool": {

5 "must": [

6 {

7 "match\_all": {}

8 }

9 ],

10 "filter": {

11 "geo\_distance": {

12 "distance": "200km",

13 "pin": {

14 "lat": 40,

15 "lon": ‐70

16 }

17 }

18 }

19 }

20 }

21 }

22

23 GET hotel\_app/\_search

24 {

25 "query": {

26 "geo\_distance" : {

27 "distance" : "90km",

28 "pin" : {

29 "lat" : 40.55,

30 "lon" : ‐71.12

31 }

32 }

33 }

34 }

5、统计某位置附近区域内的数据

聚合统计分别距离某位置80英里，300英里，1000英里范围内的数据数量。

其中unit是距离单位，常用单位有：米（m），千米（km），英里（mi）

distance\_type是统计算法：sloppy\_arc默认算法、arc最高精度、plane最高效率

1 GET /hotel\_app/\_doc/\_search

2 {

3 "size": 0,

4 "aggs": {

5 "agg\_by\_pin" : {

6 "geo\_distance": {

7 "distance\_type": "arc",

8 "field": "pin",

9 "origin": {

10 "lat": 40,

11 "lon": ‐70

12 },

13 "unit": "mi",

14 "ranges": [

15 {

16 "to": 80

17 },

18 {

19 "from": 80,

20 "to": 300

21 },

22 {

23 "from": 300,

24 "to": 1000

25 }

26 ]

27 }

28 }

29 }

30 }

#### 3、连接查询(多文档合并查询)

父子文档查询：parent/child

嵌套文档查询: nested

**父子文档优缺点：**

优点：文档存储在一起，读取性能高、父子文档可以独立更新

缺点：更新嵌套的子文档时，需要更新整个文档、需要额外的内存去维护关系。读取性能

相对差适用场景子文档偶尔更新，以查询为主、子文档更新频繁

**嵌套文档优缺点**：

嵌套文档的nested子文档在ES内部其实也是独立的lucene文档，只是我们在查询的时候，ES内部帮我们做了join处理。最终看起来好像是一个独立的文档一样。嵌套对象通过冗余数据来提高查询性能，适用于读多写少的场景，但是它有两个缺点，一个是更新主文档的时候要全部更新（冗余数据类似于根据嵌套属性生成多条单独的数据，类似于），另外就是不支持子文档从属多个主文档的场景。但相比于父子文档查询速度快一个量级

##### 3.1 父子文档

nested object的建模，有个不好的地方，就是采取的是类似冗余数据的方式，将多个数据

都放在一起了，维护成本就比较高，每次更新，需要重新索引整个对象（包括跟对象和嵌套对象）ES 提供了类似关系型数据库中 Join 的实现。使用 Join 数据类型实现，可以通过

Parent / Child 的关系，从而分离两个对象父文档和子文档是两个独立的文档，更新父文档无需重新索引整个子文档。子文档被新增，更改和删除也不会影响到父文档和其他子文档。

要点：父子关系元数据映射，用于确保查询时候的高性能，但是有一个限制，就是父子数

据必须存在于一个shard中

父子关系数据存在一个shard中，而且还有映射其关联关系的元数据，那么搜索父子关系数

据的时候，不用跨分片，一个分片本地自己就搞定了，性能当然高

父子关系

定义父子关系的几个步骤

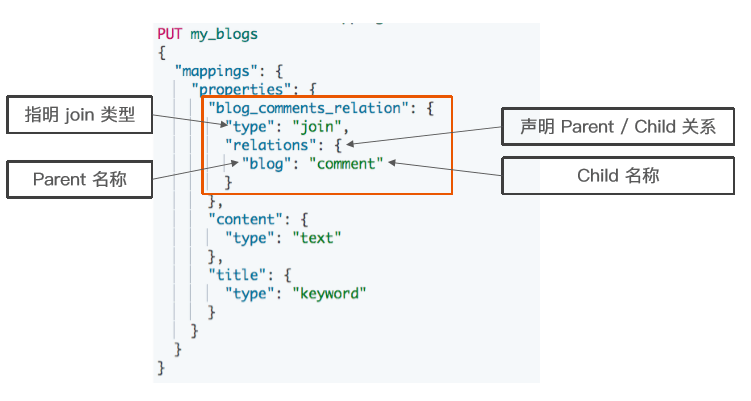
设置索引的 Mapping

索引父文档

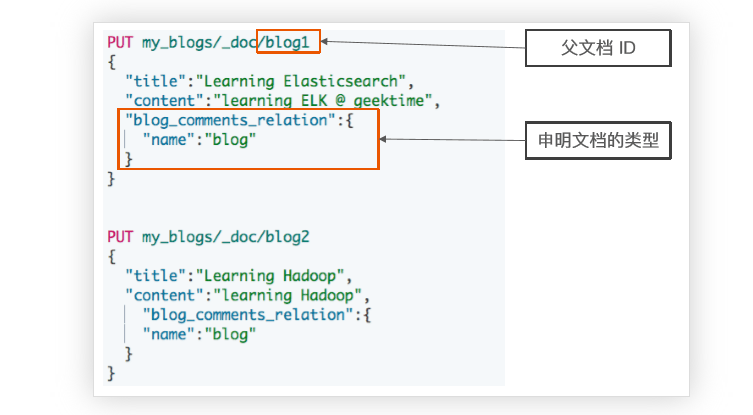
索引子文档

按需查询文档

设置 Mapping



**插入父文档数据**



PUT my\_blogs/\_doc/blog1{

"title":"Learning Elasticsearch",

"content":"learning ELK is happy",

"blog\_comments\_relation":{

"name":"blog"

}}

PUT my\_blogs/\_doc/blog2{

"title":"Learning Hadoop",

"content":"learning Hadoop",

"blog\_comments\_relation":{

"name":"blog"

}}

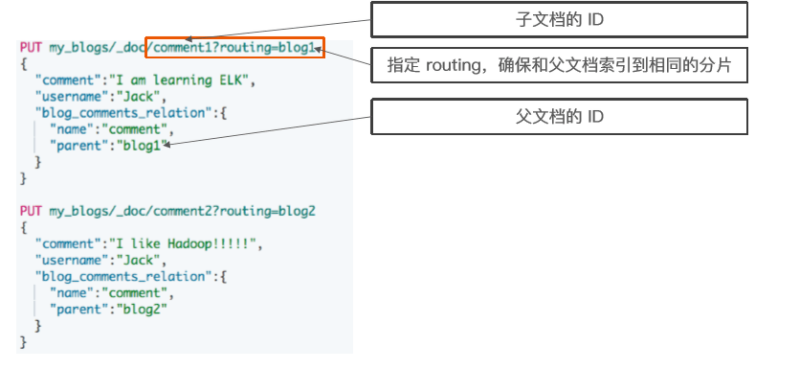
**插入子文档数据**

父文档和子文档必须存在相同的分片上

确保查询 join 的性能

当指定文档时候，必须指定它的父文档 ID

使用 route 参数来保证，分配到相同的分片



PUT my\_blogs/\_doc/comment1?routing=blog1{

"comment":"I am learning ELK",

"username":"Jack",

"blog\_comments\_relation":{

"name":"comment",

"parent":"blog1"

}}

PUT my\_blogs/\_doc/comment2?routing=blog2{

"comment":"I like Hadoop!!!!!",

"username":"Jack",

"blog\_comments\_relation":{

"name":"comment",

"parent":"blog2"

}}

PUT my\_blogs/\_doc/comment3?routing=blog2{

"comment":"Hello Hadoop",

"username":"Bob",

"blog\_comments\_relation":{

"name":"comment",

"parent":"blog2"

}}

**Parent / Child 所支持的查询**

查询所有文档

Parent Id 查询

Has Child 查询

Has Parent 查询

# 查询所有文档

POST my\_blogs/\_search

{}

#根据父文档ID查看

GET my\_blogs/\_doc/blog2

# Parent Id 查询，返回所有相关子文档

POST my\_blogs/\_search{

"query": {

"parent\_id": {

"type": "comment",

"id": "blog2"

}}}

# Has Child 查询,返回父文档

POST my\_blogs/\_search{

"query": {

"has\_child": {

"type": "comment",

"query" : {

"match": {

"username" : "Jack"

}}}}}

# Has Parent 查询，返回相关的子文档

POST my\_blogs/\_search{

"query": {

"has\_parent": {

"parent\_type": "blog",

"query" : {

"match": {

"title" : "Learning Hadoop"

}}}}}

#通过ID和routing ，访问子文档

GET my\_blogs/\_doc/comment3?routing=blog2

# 更新子文档

PUT my\_blogs/\_doc/comment3?routing=blog2{

"comment": "Hello Hadoop??",

"blog\_comments\_relation": {

"name": "comment",

"parent": "blog2"

}}

##### 3.2 嵌套文档

设计一个用户document数据类型，其中包含一个地址数据的数组，这种设计方式

相对复杂，但是在管理数据时，更加的灵活。

PUT /user\_index{

"mappings": {

"properties": {

"login\_name" : {

"type" : "keyword"

},

"age " : {

"type" : "short"

},

"address" : {

"properties": {

"province" : {

"type" : "keyword"

},

"city" : {

"type" : "keyword"

},

"street" : {

"type" : "keyword"

}}}}}}

但是上述的数据建模有其明显的缺陷，就是针对地址数据做数据搜索的时候，经常会搜索出不必要的数据，其在创建倒排索引的时候字段province{北京，天津}，city{北京，天津}如：在下述数据环境中，搜索一个province为北京，city为天津的用户，搜索时只要一条数据中province与city的索引数组中包含北京、天津即可被查询出，但实际上不存在北京省天津市的数据。

PUT /user\_index/\_doc/1{

"login\_name" : "jack",

"age" : 25,

"address" : [{

"province" : "北京",

"city" : "北京",

"street" : "枫林三路"

},

{

"province" : "天津",

"city" : "天津",

"street" : "华夏路"

}]}

PUT /user\_index/\_doc/2{

"login\_name" : "rose",

"age" : 21,

"address" : [{

"province" : "河北",

"city" : "廊坊",

"street" : "燕郊经济开发区"

},

{

"province" : "天津",

"city" : "天津",

"street" : "华夏路"

}]}

执行的搜索应该如下：

GET /user\_index/\_search{

"query": {

"bool": {

"must": [{

"match": {

"address.province": "北京"

}},

{

"match": {

"address.city": "天津"

}}]}}}

但是得到的结果并不准确，这个时候就需要使用nested object来定义数据建模。

使用nested object作为地址数组的集体类型，可以解决上述问题，document模型如下：

PUT /user\_index{

"mappings": {

"properties": {

"login\_name" : {

"type" : "keyword"

},

"age" : {

"type" : "short"

},

"address" : {

"type": "nested",

"properties": {

"province" : {

"type" : "keyword"

},

"city" : {

"type" : "keyword"

},

"street" : {

"type" : "keyword"

}}}}}}

这个时候就需要使用nested对应的搜索语法来执行搜索了，语法如下：

GET /user\_index/\_search{

"query": {

"bool": {

"must": [{

"nested": {

"path": "address",

"query": {

"bool": {

"must": [{

"match": {

"address.province": "北京"

}},

{

"match": {

"address.city": "天津"

}}]}}}}]}}}

虽然语法变的复杂了，但是在数据的读写操作上都不会有错误发生，是推荐的设计方式。

其原因是：普通的数组数据在ES中会被扁平化处理，处理方式如下：（如果字段需要分词，会将分词数据保存在对应的字段位置，当然应该是一个倒排索引，这里只是一个直观的案例）

{

  "login\_name" : "jack",

  "address.province" : [ "北京", "天津" ],

  "address.city" : [ "北京", "天津" ]

  "address.street" : [ "西三旗东路", "古文化街" ]

}

那么nested object数据类型ES在保存的时候不会有扁平化处理，保存方式如下：所以在搜

索的时候一定会有需要的搜索结果。

{

  "login\_name" : "jack"

}

{

  "address.province" : "北京",

  "address.city" : "北京"，

  "address.street" : "西三旗东路"

}

{

  "address.province" : "北京",

  "address.city" : "北京",

  "address.street" : "西三旗东路",

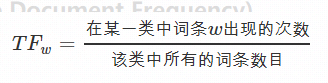
}

GET：term为精确匹配，match为模糊查询，query\_string

创建索引的时候，因为分析器的原因，大写的英文数据如DOG会自动变为小写的的dog，match与query\_string查询时内容也会被分析会自动判断查询为dog，而term会直接查询DOG，导致搜索不到

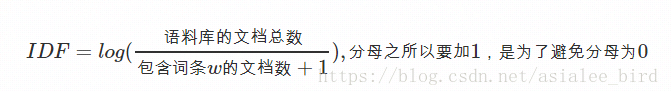
TF-IDF：

tf: ：词频这个document文档包含了多少个这个词n，包含越多表明越相关



df：有m个文档包含了这个词；这个词所有的文档都有，那这个就没有意义。

idf: ：逆文档频率1/m， 包含该词的文档总数目，idf值越大即m越小，则该词对文档的区分度越好

 TF-IDF：tf\*idf= n/m

tfNom: : 根据field长度做归一化，文档内出现频率越高，field越短越相关操作不管是字符与还是或，按照逻辑关系命中后相加得分

### 聚合查询

**1.bucket和metric概念简介**

bucket就是一个聚合搜索时的数据分组。如：销售部门有员工张三和李四，开发部门有员工王五和赵六。那么根据部门分组聚合得到结果就是两个bucket。销售部门bucket中有张三和李四，开发部门 bucket中有王五和赵六。

metric就是对一个bucket数据执行的统计分析。如上述案例中，开发部门有2个员

工，销售部门有2个员工，这就是metric。

metric有多种统计，如：求和，最大值，最小值，平均值等。

1 用一个大家容易理解的SQL语法来解释，如：select count(\*) from table group by colum

n。那么group by column分组后的每组数据就是bucket。对每个分组执行的count(\*)就是metric。

PUT /cars{

"mappings": {

"properties": {

"price": {

"type": "long"

},

"color": {

"type": "keyword"

},

"brand": {

"type": "keyword"

},

"model": {

"type": "keyword"

},

"sold\_date": {

"type": "date"

},

"remark" : {

"type" : "text",

"analyzer" : "ik\_max\_word"

}}}}

POST /cars/\_bulk

{ "index": {}}

{ "price" : 258000, "color" : "金色", "brand":"大众", "model" : "大众迈腾", "sol

d\_date" : "2021‐10‐28","remark" : "大众中档车" }

4 { "index": {}}

5 { "price" : 123000, "color" : "金色", "brand":"大众", "model" : "大众速腾", "sol

d\_date" : "2021‐11‐05","remark" : "大众神车" }

6 { "index": {}}

7 { "price" : 239800, "color" : "白色", "brand":"标志", "model" : "标志508", "sold

\_date" : "2021‐05‐18","remark" : "标志品牌全球上市车型" }

8 { "index": {}}

9 { "price" : 148800, "color" : "白色", "brand":"标志", "model" : "标志408", "sold

\_date" : "2021‐07‐02","remark" : "比较大的紧凑型车" }

10 { "index": {}}

11 { "price" : 1998000, "color" : "黑色", "brand":"大众", "model" : "大众辉腾", "so

ld\_date" : "2021‐08‐19","remark" : "大众最让人肝疼的车" }

12 { "index": {}}

13 { "price" : 218000, "color" : "红色", "brand":"奥迪", "model" : "奥迪A4", "sold

\_date" : "2021‐11‐05","remark" : "小资车型" }

14 { "index": {}}

15 { "price" : 489000, "color" : "黑色", "brand":"奥迪", "model" : "奥迪A6", "sold

\_date" : "2022‐01‐01","remark" : "政府专用？" }

16 { "index": {}}

17 { "price" : 1899000, "color" : "黑色", "brand":"奥迪", "model" : "奥迪A 8", "so

ld\_date" : "2022‐02‐12","remark" : "很贵的大A6。。。" }

#### 1、根据color分组统计销售数量

只执行聚合分组，不做复杂的聚合统计。在ES中最基础的聚合为terms，相当于

SQL中的count。

在ES中默认为分组数据做排序，使用的是doc\_count数据执行降序排列。可以使用

\_key元数据，根据分组后的字段数据执行不同的排序方案，也可以根据\_count元数

据，根据分组后的统计值执行不同的排序方案。

GET /cars/\_search{

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color",

"order": {

"\_count": "desc"

}}}}}

#### 2、统计不同color车辆的平均价格

本案例先根据color执行聚合分组，在此分组的基础上，对组内数据执行聚合统计，这个组内数据的聚合统计就是metric。同样可以执行排序，因为组内有聚合统计，且对统计数据给予了命名avg\_by\_price，所以可以根据这个聚合统计数据字段名执行排序逻辑。

GET /cars/\_search{

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color",

"order": {

"avg\_by\_price": "asc"

}},

"aggs": {

"avg\_by\_price": {

"avg": {

"field": "price"

}}}}}}

size可以设置为0，表示不返回ES中的文档，只返回ES聚合之后的数据，提高查询速

度，当然如果你需要这些文档的话，也可以按照实际情况进行设置

GET /cars/\_search{

"size" : 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"group\_by\_brand" : {

"terms": {

"field": "brand",

"order": {

"avg\_by\_price": "desc"

}},

"aggs": {

"avg\_by\_price": {

"avg": {

"field": "price"

}}}}}}}}

#### 3、统计不同color不同brand中车辆的平均价格

先根据color聚合分组，在组内根据brand再次聚合分组，这种操作可以称为下钻

分析。

Aggs如果定义比较多，则会感觉语法格式混乱，aggs语法格式，有一个相对固定

的结构，简单定义：aggs可以嵌套定义，可以水平定义。

嵌套定义称为下钻分析。水平定义就是平铺多个分组方式。

GET /index\_name/type\_name/\_search{

"aggs" : {

"定义分组名称（最外层）": {

"分组策略如：terms、avg、sum" : {

"field" : "根据哪一个字段分组",

"其他参数" : ""

},

"aggs" : {

"分组名称1" : {},

"分组名称2" : {}

}}}}

GET /cars/\_search{

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color",

"order": {

"avg\_by\_price\_color": "asc"

}},

"aggs": {

"avg\_by\_price\_color" : {

"avg": {

"field": "price"

}},

"group\_by\_brand" : {

"terms": {

"field": "brand",

"order": {

"avg\_by\_price\_brand": "desc"

}},

"aggs": {

"avg\_by\_price\_brand": {

"avg": {

"field": "price"

}}}}}}}}

#### 4、统计不同color中的最大和最小价格、总价

GET /cars/\_search{

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"max\_price": {

"max": {

"field": "price"

}},

"min\_price" : {

"min": {

"field": "price"

}},

"sum\_price" : {

"sum": {

"field": "price"

}}}}}}

在常见的业务常见中，聚合分析，最常用的种类就是统计数量，最大，最小，平均，

总计等。通常占有聚合业务中的60%以上的比例，小型项目中，甚至占比85%以上。

#### 5、统计不同品牌汽车中价格排名最高的车型

在分组后，可能需要对组内的数据进行排序，并选择其中排名高的数据。那么可

以使用s来实现：top\_hit,hits中的属性size代表取组内多少条数据（默认为

10）；sort代表组内使用什么字段什么规则排序（默认使用\_doc的asc规则排序）；

\_source代表结果中包含document中的那些字段（默认包含全部字段）。

GET cars/\_search{

"size" : 0,

"aggs": {

"group\_by\_brand": {

"terms": {

"field": "brand"

},

"aggs": {

"top\_car": {

"top\_hits": {

"size": 1,

"sort": [

{

"price": {

"order": "desc"

}}],

"\_source": {

"includes": ["model", "price"]

}}}}}}}

#### 6、histogram 区间统计

histogram类似terms，也是进行bucket分组操作的，是根据一个field，实现数据

区间分组。

如：以100万为一个范围，统计不同范围内车辆的销售量和平均价格。那么使用

histogram的聚合的时候，field指定价格字段price。区间范围是100万-interval ：

1000000。这个时候ES会将price价格区间划分为： [0, 1000000), [1000000,

2000000), [2000000, 3000000)等，依次类推。在划分区间的同时，histogram会类似

terms进行数据数量的统计（count），可以通过嵌套aggs对聚合分组后的组内数据做

再次聚合分析。

GET /cars/\_search{

"aggs": {

"histogram\_by\_price": {

"histogram": {

"field": "price",

"interval": 1000000

},

"aggs": {

"avg\_by\_price": {

"avg": {

"field": "price"

}}}}}}

#### 7、date\_histogram区间分组

date\_histogram可以对date类型的field执行区间聚合分组，如每月销量，每年销量等。

如：以月为单位，统计不同月份汽车的销售数量及销售总金额。这个时候可以使用date\_histogram实现聚合分组，其中field来指定用于聚合分组的字段，interval指定区间范围（可选值有：year、quarter、month、week、day、hour、minute、second），format指定日期格式化，min\_doc\_count指定每个区间的最少document（如果不指定，默认为0，当区间范围内没有document时，也会显示bucket分组），extended\_bounds指定起始时间和结束时间（如果不指定，默认使用字段中日期最小值所在范围和最大值所在范围为起始和结束时间）。

GET /cars/\_search{

"aggs": {

"histogram\_by\_date" : {

"date\_histogram": {

"field": "sold\_date",

"calendar\_interval": "month",

"format": "yyyy‐MM‐dd",

"min\_doc\_count": 1,

"extended\_bounds": {

"min": "2021‐01‐01",

"max": "2022‐12‐31"

}},

"aggs": {

"sum\_by\_price": {

"sum": {

"field": "price"

}}}}}}

#### 8、global bucket

在聚合统计数据的时候，有些时候需要对比部分数据和总体数据。

如：统计某品牌车辆平均价格和所有车辆平均价格。global是用于定义一个全局bucket，这个bucket会忽略query的条件，检索所有document进行对应的聚合统计。

GET /cars/\_search{

"size" : 0,

"query": {

"match": {

"brand": "大众"

}},

"aggs": {

"volkswagen\_of\_avg\_price": {

"avg": {

"field": "price"

}},

"all\_avg\_price" : {

"global": {},

"aggs": {

"all\_of\_price": {

"avg": {

"field": "price"

}}}}}}

#### 9、aggs+order

对聚合统计数据进行排序。

如：统计每个品牌的汽车销量和销售总额，按照销售总额的降序排列。

GET /cars/\_search{

"aggs": {

"group\_of\_brand": {

"terms": {

"field": "brand",

"order": {

"sum\_of\_price": "desc"

}},

"aggs": {

"sum\_of\_price": {

"sum": {

"field": "price"

}}}}}}

如果有多层aggs，执行下钻聚合的时候，也可以根据最内层聚合数据执行排序。

如：统计每个品牌中每种颜色车辆的销售总额，并根据销售总额降序排列。这就像

SQL中的分组排序一样，只能组内数据排序，而不能跨组实现排序。

GET /cars/\_search{

"aggs": {

"group\_by\_brand": {

"terms": {

"field": "brand"

},

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color",

"order": {

"sum\_of\_price": "desc"

}},

"aggs": {

"sum\_of\_price": {

"sum": {

"field": "price"

}}}}}}}}

#### 10、search+aggs

聚合类似SQL中的group by子句，search类似SQL中的where子句。在ES中是完全可以将search和aggregations整合起来，执行相对更复杂的搜索统计。

如：统计某品牌车辆每个季度的销量和销售额。

GET /cars/\_search{

"query": {

"match": {

"brand": "大众"

}},

"aggs": {

"histogram\_by\_date": {

"date\_histogram": {

"field": "sold\_date",

"calendar\_interval": "quarter",

"min\_doc\_count": 1

},

"aggs": {

"sum\_by\_price": {

"sum": {

"field": "price"

}}}}}}

#### 11、filter+aggs

在ES中，filter也可以和aggs组合使用，实现相对复杂的过滤聚合分析。

如：统计10万~50万之间的车辆的平均价格。

GET /cars/\_search{

"query": {

"constant\_score": {

"filter": {

"range": {

"price": {

"gte": 100000,

"lte": 500000

}}}}},

"aggs": {

"avg\_by\_price": {

"avg": {

"field": "price"

}}}}

#### 12、聚合中使用filter

filter也可以使用在aggs句法中，filter的范围决定了其过滤的范围。

如：统计某品牌汽车最近一年的销售总额。将filter放在aggs内部，代表这个过滤器只对query搜索得到的结果执行filter过滤。如果filter放在aggs外部，过滤器则会过滤所有的数据。

12M/M 表示 12 个月。

1y/y 表示 1年。

d 表示天

GET /cars/\_search{

"query": {

"match": {

"brand": "大众"

}},

"aggs": {

"count\_last\_year": {

"filter": {

"range": {

"sold\_date": {

"gte": "now‐12M"

}}},

"aggs": {

"sum\_of\_price\_last\_year": {

"sum": {

"field": "price"

}}}}}}

### 分数计算原理

ElasticSearch文档分值\_score计算底层原理

**1、boolean model**

根据用户的query条件，先过滤出包含指定term的doc

query "hello world" ‐‐> hello / world / hello & world

bool ‐‐> must/must not/should ‐‐> 过滤 ‐‐> 包含 / 不包含 / 可能包含

doc ‐‐> 不打分数 ‐‐> 正或反 true or false ‐‐> 为了减少后续要计算的doc的数量，提升性

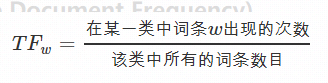
能

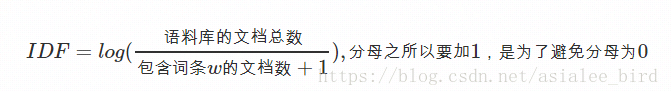
**2、relevance score算法，简单来说，就是计算出，一个索引中的文本，与搜索文本，**

**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

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

搜索请求：hello world

doc1：hello, Kevin 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

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": "...... N个单词" }

doc2：{ "title": "my article", "content": "...... N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

**3、vector space model**

多个term对一个doc的总分数

hello world --> es会根据hello world在所有doc中的评分情况，计算出一个query vector，query向量

hello这个term，给的基于所有doc的一个评分就是2

world这个term，给的基于所有doc的一个评分就是5

[2, 5]

query vector

doc vector，3个doc，一个包含1个term，一个包含另一个term，一个包含2个term

3个doc

doc1：包含hello --> [2, 0]

doc2：包含world --> [0, 5]

doc3：包含hello, world --> [2, 5]

会给每一个doc，拿每个term计算出一个分数来，hello有一个分数，world有一个分数，再拿所有term的分数组成一个doc vector

画在一个图中，取每个doc vector对query vector的弧度，给出每个doc对多个term的总分数

每个doc vector计算出对query vector的弧度，最后基于这个弧度给出一个doc相对于query中多个term的总分数，弧度越大，分数月底; 弧度越小，分数越高

如果是多个term，那么就是线性代数来计算，无法用图表示

### 分词器

1. stander中文的默认分词，这是按照一个个字分开。此种分词非常占空间会导致倒排索引很大，而且在搜索的时候也会搜出很多不相干的东西。优点就是搜的多。
2. English分词器：会提取词干和去掉停用词，并将大写转为小写

3、ik分词器

ik配置文件地址：es/plugins/ik/config目录

IKAnalyzer.cfg.xml：用来配置自定义词库

main.dic：ik原生内置的中文词库，总共有27万多条，只要是这些单词，都会被分在一起

quantifier.dic：放了一些单位相关的词

suffix.dic：放了一些后缀

surname.dic：中国的姓氏

stopword.dic：英文停用词

ik原生最重要的两个配置文件

main.dic：包含了原生的中文词语，会按照这个里面的词语去分词

stopword.dic：包含了英文的停用词

停用词，stopword：

a the and at but一般，像停用词，会在分词的时候，直接被干掉，不会建立在倒排索引中

**IK分词器自定义词库**

（1）自己建立词库：每年都会涌现一些特殊的流行词，网红，蓝瘦香菇，喊麦，鬼

畜，一般不会在ik的原生词典里自己补充自己的最新的词语，到ik的词库里面去

IKAnalyzer.cfg.xml：ext\_dict，custom/mydict.dic

补充自己的词语，然后需要重启es，才能生效

（2）自己建立停用词库：比如了，的，啥，么，我们可能并不想去建立索引，让人家

搜索custom/ext\_stopword.dic，已经有了常用的中文停用词，可以补充自己的停用词，然

后重启es

**IK分词器源码下载：https://github.com/medcl/elasticsearch‐analysis‐ik/tree**

**IK热更新**

每次都是在es的扩展词典中，手动添加新词语，很坑

（1）每次添加完，都要重启es才能生效，非常麻烦

（2）es是分布式的，可能有数百个节点，你不能每次都一个一个节点上面去修改

es不停机，直接我们在外部某个地方添加新的词语，es中立即热加载到这些新词语

IKAnalyzer.cfg.xml

<properties>

<comment>IK Analyzer 扩展配置</comment>

<!‐‐用户可以在这里配置自己的扩展字典 ‐‐>

<entry key="ext\_dict">location</entry>

<!‐‐用户可以在这里配置自己的扩展停止词字典‐‐>

<entry key="ext\_stopwords">location</entry>

<!--用户可以在这里配置远程扩展字典 -->

<entry key="remote\_ext\_dict">words\_location</entry>

<!--用户可以在这里配置远程扩展停止词字典-->

<entry key="remote\_ext\_stopwords">words\_location</entry>

</properties>

### 1.5 es SQL

Elasticsearch SQL允许执行类SQL的查询，可以使用REST接口、命令行或者是JDBC，都可

以使用SQL来进行数据的检索和数据的聚合。

Elasticsearch SQL特点：

本地集成，Elasticsearch SQL是专门为Elasticsearch构建的。每个SQL查询都根据底层存储对相关节点有效执行。没有额外的要求不依赖其他的硬件、进程、运行时库，Elasticsearch SQL可以直接运行在Elasticsearch集群上，轻量且高效，像SQL那样简洁、高效地完成查询

**SQL与Elasticsearch对应关系**



**Elasticsearch SQL语法**

SELECT select\_expr [, ...]

[ FROM table\_name ]

[ WHERE condition ]

[ GROUP BY grouping\_element [, ...] ]

[ HAVING condition]

[ ORDER BY expression [ ASC | DESC ] [, ...] ]

[ LIMIT [ count ] ]

[ PIVOT ( aggregation\_expr FOR column IN ( value [ [ AS ] alias ] [, ...] ) ) ]

目前FROM只支持单表

**职位查询案例**

查询职位索引库中的一条数据

format：表示指定返回的数据类型

1.查询职位信息

GET /\_sql?format=txt{

"query":"SELECT \* FROM es\_db limit 1"

}

除了txt类型，Elasticsearch SQL还支持以下类型，

格式 描述

csv 逗号分隔符

json JSON格式

tsv 制表符分隔符

txt 类cli表示

yaml YAML人类可读的格式

**将SQL转换为DSL**

GET /\_sql/translate{

3 "query":"SELECT \* FROM es\_db limit 1"

4 }

结果如下：

1 {

2  "size" : 1,

3  "\_source" : {

4  "includes" : [

5  "age",

6  "remark",

7  "sex"

8  ],

9  "excludes" : [ ]

10  },

11  "docvalue\_fields" : [

12  {

13  "field" : "address"

14  },

15  {

16  "field" : "book"

17  },

18  {

19  "field" : "name"

20  }

21  ],

22  "sort" : [

23  {

24  "\_doc" : {

25  "order" : "asc"

26  }

27  }

28  ]

29 }

3.4、职位全文检索

3.4.1、需求

检索address包含广州和name中包含张三的用户。

3.4.2、MATCH函数

在执行全文检索时，需要使用到MATCH函数。

1 MATCH(

2     field\_exp,

3     constant\_exp

4     [, options])

field\_exp：匹配字段

constant\_exp：匹配常量表达式

3.4.3、实现

1 GET /\_sql?format=txt

2 {

3 "query":"select \* from es\_db where MATCH(address, '广州') or MATCH(name, '张三') limi

t 10"

4 }

4.4、通过 Elasticsearch SQL方式实现分组统计

4.4.2、基于Elasticsearch SQL方式实现

1 GET /\_sql?format=txt

2 {

3 "query":"select age, count(\*) as age\_cnt from es\_db group by age"

4 }

这种方式要更加直观、简洁。

Elasticsearch SQL目前的一些限制

目前Elasticsearch SQL还存在一些限制。例如：不支持JOIN、不支持较复杂的子查询。所

以，有一些相对复杂一些的功能，还得借助于DSL方式来实现。

### 1 term精确匹配

GET /book/\_search

{

"query": {

"term": {

"bookName": {

"value": "童话故事大全"

}

}

}

Term查询值使用value，match查询值使用query

Terms查询可以使用include包含，exclude排除

POST /nba/\_search

{

"aggs": {

"aggsTeamName": {

"terms": {

"field": "teamNameEn",

"include": [

"Lakers",

"Rockets",

"Warriors"

],

"exclude": [

"Warriors"

],

"size": 30,

"order": {

"avgAge": "desc"

}

},

"aggs": {

"avgAge": {

"avg": {

"field": "age"

}

}

}

}

},

"size": 0

}

在my\_index 索引中查找与文档id为2的color字段有共同值得选项

GET my\_index/\_search?pretty

{

"query": {

"terms": {

"color" : {

"index" : "my\_index",

"id" : "2",

"path" : "color"

}

}

}

}

### 2 短语查询

Match\_phase:查询确切的phase即搜索的值，例如hello world，返回满足如下两个条件的数据，1、match\_phase中所有的term都出现在待查字段中 2、待查字段中所有term都必须和match\_phase具有相同的顺序

match\_phrase\_prefix:可实现即时的短语匹配查询，与match\_phrase查询相同，不同的是它将查询字符串的最后一个词作为前缀使用，即查询时先匹配最后一个词

"match\_phrase\_prefix" : {

"brand" : {

"query": "walker johnnie bl", (1)

"slop": 10，

"max\_expansions": 50

}

}

Slop设置为使搜索的字符串各个词的顺序匹配不那么严格

max\_expansions：prefix 查询存在严重的资源消耗问题，短语查询的这种方式也同样如此。前缀 a 可能会匹配成千上万的词，这不仅会消耗很多系统资源，而且结果的用处也不大。

可以通过设置 max\_expansions 参数来限制前缀扩展的影响，一个合理的值是可能是 50 ，参数 max\_expansions 控制着可以与前缀匹配的词的数量，它会先查找第一个与前缀 bl 匹配的词，然后依次查找搜集与之匹配的词（按字母顺序），直到没有更多可匹配的词或当数量超过 max\_expansions 时结束。

### 3.多字段查询

Multi\_match：多字段查询

GET /book/\_search

{

"query": {

"multi\_match": {

"query": "大自然的旅行故事",

"fields": ["bookName","discription"]

}

}

}

此时为默认best\_fields类型，即取两边分词分数高的那个字段，多字段查询中如果字段中匹配到多个分词，则此字段分数为多个分词分数之和

字段加权重，即在bookName字段的分数乘10

"fields": ["bookName^10","discription"]

添加tie\_breaker：

"tie\_breaker": 0.3 放在fields下面

表示单一分词在两个字段中的得分取最大值加上其余值的0.3倍

（单一字段下所有分词查询取最大值加上其余分词的总分数乘以tie\_breaker）

多字段分值相加：

"type": "most\_fields" 即该条数据的总分数为多个字段分数之和

以分词为单位分别在每个字段里面得分取最大相加，非常适用于以词作为权重的系统：

"type": "cross\_fields"，以分词为单位，单个分词的得分取该分词在所有字段中的最大值，然后多个分词的得分之和为该条数据的总得分

### 4.排序、分页

查询所有

GET /test/\_search

或者

GET /test/\_search

{

"query":{

"match\_all": {}

}

}

分页查询

GET /test/\_search

{

"query":{

"match\_all": {}

},

"from":0,

"size":2

}

带条件查询

GET /test/\_search

{

"query": {

"match": {

"name": "kevin"

}

}

}

### 5.最小匹配查询

GET /book/\_analyze

{

"field": "bookName",

"text": "安徒生的大自然童话故事"

}

GET /book/\_search

{

"query": {

"match": {

"bookName": {

"query": "安徒生的大自然童话故事",

"operator": "or",

"minimum\_should\_match": 2

}

}

}

}

"operator": "or"，or代表搜索的字符串所分的词用于检索式有一个即可，and代表需要包含所有分词

"minimum\_should\_match": 2 必须包含的分词个数，用于控制返回结果数，此时operator需要为or

### 6、analyze、分词器与validate-query

**analyze api** 根据所选择的分词器将text数据进行分词操作

GET /test/\_analyze

{

"field": "name",

"text": "my name is zhaoyun and i like eating apples and running"

}

从上面的结果中我们可以看出es对其进行了分词。Es中主要有三种分词：

(1)stander中文的默认分词，这是按照一个个字分开。此种分词非常占空间会导致倒排索引很大，而且在搜索的时候也会搜出很多不相干的东西。优点就是搜的多。

(2)English分词器：会提取词干和去掉停用词，并将大写转为小写。假设name不采用英文分词：

PUT /test/\_doc/1

{

"name": "eating apples,

"age": 30

}

然后搜索

GET /test/\_search

{

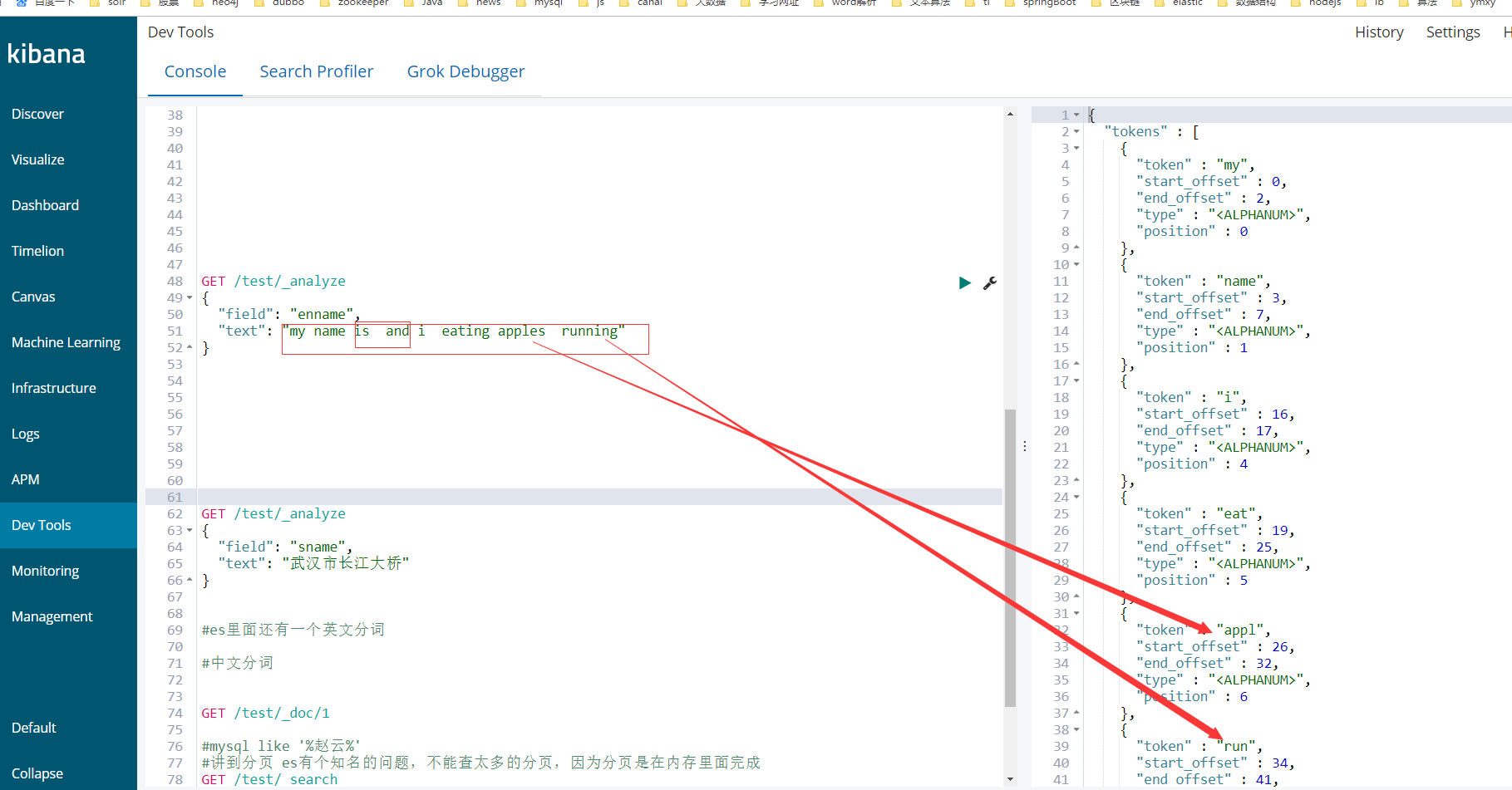
"query":{

"match": {"name":"eat"}

}

}

没搜到结果，我们使用后使用analyze api查看分析处理结果，可以看到没有分出eat，所以搜不到，改成用english分词器做。就可以看到以下的情况



可以看到running和apples被提取成了词干，提取算法其实就是一个映射感兴趣的同学可以自己取看看。而and 和 is被当作停用词去掉了。

（3）IK分词器：他会有两种模式一种smart，一种max\_word. 建索引的时候 用ik\_max\_word，搜索的时候用smart（查询时使用尽量长的分词）。如果需要详细理解里面的原理的，请看算法课的第23节课。讲的相当详细。

Ik分词器某个分词在存储包含该分词的文档id时使用的链表来保存，热点分词的链表可能很长，所以会进行压缩，查找会使用跳表进行查询

**分词的妙用：**

是不是就意味着stander分词没用了？其实并不是这样的哦

#托底，搜江大桥没有，在建了ik的字段，在建一个一样的stander的字段。如果ik搜不到 就可以搜这一个stander分词的，这样保证会有结果。但是慎用，因为占空间，有些特殊的系统可以使用。

#其实ik也还有一个解决的办法 叫砍词：江大桥 我可以砍掉一个词，我砍掉江 就出来了。砍词的策略可以自定义

#江大桥：电商中。我们系统假设有大桥这个品牌。 我会一个个的是去试一下，比如可以用字符串匹配找出大桥。也有很多系统很粗暴，直接从第一个字开始砍，一直砍到有为止。

#既有英文又有中文的 直接选ik

#如果不用砍词那就要去词库加词，比如加入江大桥就可以了。具体路径是在es的：

/apps/svr/elasticsearch-node1/config/analysis-ik/main.dic

注意集群的话那就要所有的es都需要加的哦

验证查询validate-query：

GET /book/\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "童话故事的大选",

"fields": ["bookName","discription"]

}

}

}

当不加？explain时，返回查询是否合法的信息，当想了解具体原因时需要加上

也可以最直接在query上面加上"explain":true,

GET /book/\_search

{

"explain": true,

"query": {

"multi\_match": {

"query": "大自然的旅行故事",

"fields": ["bookName","discription"],

"type": "cross\_fields"

}

}

}

### 7、queryString

GET /book/\_search

{

"query": {

"query\_string": {

"fields": ["bookName"],

"query": "大自然 AND 旅行"

}

}

}

default\_field： 查询的字段，支持多字段查询"fields": ["bookName"，”description”]

default\_operator:AND代表包含两个分词，ＯＲ为或，必须大写,or为默认值，可以复合使用”(this AND that) OR thu\*”

### ８、bool过滤器

Bool过滤器由三部分组成：

Must: 所有语句都必须匹配，等价于and

Must\_not: 所有语句都不能匹配，与not等价

Should：至少有一个语句匹配，与or等价

Should与 must\_not 混合使用，should的两个条件满足一个即可，并且必须满足must\_not

GET /test/\_search

{

"query": {

"bool": {

"should": [

{

"term": {

"name": "java"

}

},

{

"term": {

"sname": "develop"

}

}

],

"must\_not": {

"term": {

"name": "java develop"

}

}

}

}

}

在should中嵌套一个bool查询，即bool的内部第一个条件或者嵌套bool为true都能查出来

GET /test/\_search

{

"query" : {

"bool" : {

"should" : [

{ "term" : {"name" : "java"}},

{ "bool" : {

"must" : [

{ "term" : {"sname": "develop"}},

{ "term" : {"age" : 23}}

]

}}

]

}

}

}

当bool查询为should条件时，下方的filter过滤should查出的数据，此部分数据有分数，filter仍然会过滤其他不符合should的数据，此部分数据分值为0，如果should没有结果返回，filter仍会执行所得结果分数为0

GET /test/\_search

{

"query" : {

"bool" : {

"should" : [

{ "term" : {"name" : "java"}},

{ "bool" : {

"must" : [

{ "term" : {"sname": "develop"}},

{ "term" : {"age" : 23}}

]

}}

]

,

"filter": {

"range":{

"age":{

"lte":34,

"gte":20

}

}

}

}

}

}

Filter查询是没有分数的：使用order过滤，避免没有分数

GET /book/\_search

{

"query": {

"bool": {

"filter": [

{

"range": {

"commentNum": {

"lte": 2000,

"gte": 1

}

}

},

{

"term": {

"author":"朱奎"

}

}

]

}

} ,

"sort": [

{

"commentNum": {

"order": "desc"

}

}

]

}

### 9、Function Score 自定义评分函数

score\_mode:指定如何计算组合得分是（即多个函数之间的分数计算）

multiply： 分数相乘（默认）即每个函数得分相乘

sum：分数相加

avg：分数是平均值

first：具有匹配过滤器的第一个函数被应用

max：使用最高分

min：使用最低分

max\_boost: 将新分数限定为不高于某值，如果大于则赋值为限定值

min\_score：将打分低于此值的数据过滤掉

boost\_mode:代表查询的原有得分（即主查询得分）

multiply：查询分数与功能分数（score\_mode）相乘（默认）

replace：仅使用功能分数，查询分数将被忽略

sum：查询分数与功能分数相加

avg：平均分

max：查询分数与功能分数的最大值

min：查询分数与功能分数的最小值

GET /\_search

{

"query": {

"function\_score": {

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

"boost": "5",

"functions": [

{

"filter": { "match": { "test": "bar" } },

"random\_score": {},

"weight": 23

},

{

"filter": { "match": { "test": "cat" } },

"weight": 42

}

],

"max\_boost": 42,

"score\_mode": "max",

"boost\_mode": "multiply",

"min\_score" : 42

}

}

}

**Es中预定义了一些函数，weight，field\_value\_factor,random\_score,衰减函数 linear、exp、gauss，script\_score**

Weight：权重值，单个函数得出的结果乘以权重值为当前函数的得分

**random\_score：**在相同的分片你片中，如果字段值相同，则得到的分值相同，所以字段最好选择唯一性字段，可以使用\_seq\_no字段，但是该字段在你数据更新时会变动，加入seed种子值可以使每次访问得到相同的分值

"random\_score": {

"seed": 10,

"field":"\_seq\_no"

}

**field\_value\_factor：主查询分值乘以该函数返回值**

factor: 分数乘以相应的倍数

modifier：

none：不使用任何算法公式（默认）

log：常用对数公式

log1p：给字段的值加1，然后使用普通的对数公式，加1使其更加服从高斯分布，避免重复值问题

log2p：给字段的值加2，然后使用普通的对数公式

ln：natural logarithm 自然对数，以e为底的对数

ln1p：给字段的值加1，然后使用自然对数公式

ln2p：给字段的值加2，然后使用自然对数公式

square：平方值

sqrt：开平方

reciprocal：取倒数

missing：如果文档中不包含此字段，则此字段的初始值为missing值

GET /book/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "大自然的旅行故事",

"fields": ["bookName","discription"],

"operator": "or",

"type": "most\_fields"

}

},

"functions": [

{

"field\_value\_factor": {

"field": "commentNum",

"modifier": "log2p",

"factor": 8

“missing”:1

}

}

**script\_score:使用自定义的脚本实现控制评分计算**

script中使用的字段需要为keyword类型，text类型需要将其fielddata=true

如果要忽略主查询得分，可使用boost\_mode：replace

GET /\_search

{

"query": {

"function\_score": {

"query": {

"match": { "message": "elasticsearch" }

},

"script\_score" : {

"script" : {

"params": {

"a": 5,

"b": 1.2

},

"source": "params.a / Math.pow(params.b, doc['name'].value)"

}

}

}

}

}

],

"score\_mode": "sum",

"boost\_mode": "sum"

}

}

}

Decay functions：衰减函数

有三种衰减函数—— linear 、 exp 和 gauss （线性、指数和高斯函数），它们可以操作数值、时间以及经纬度地理坐标点这样的字段。所有三个函数都能接受以下参数

Origin：*中心点* 或字段可能的最佳值，落在原点 origin 上的文档评分 \_score 为满分 1.0

Scale：衰减率，即一个文档从原点 origin 下落时，评分 \_score 改变的速度。（例如，每 £10 欧元或每 100 米）。

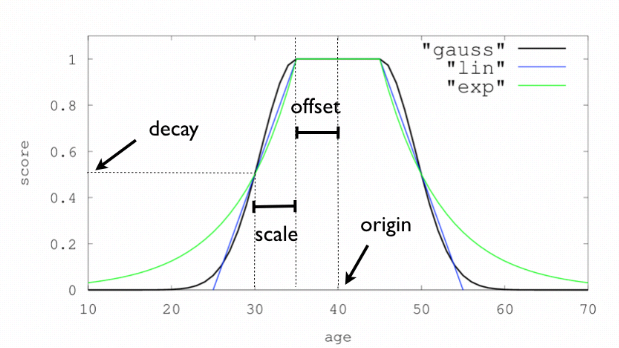
Decay：从原点 origin 衰减到 scale 所得的评分 \_score ，默认值为 0.5

Offset：以原点 origin 为中心点，为其设置一个非零的偏移量 offset 覆盖一个范围，而不只是单个原点。在范围 origin +/- offset 内的所有评分 \_score 都是 1.0

Liner:  线性函数是条直线，一旦直线与横轴 0 相交，所有其他值的评分都是 0.0

Exp:指数函数先剧烈衰减然后变缓

Gauss： 高斯函数是钟形的——它的衰减速率是先缓慢，然后变快，最后又放缓



### 10、通配符、正则、模糊查询

1、通配符

Query的内容支持？与\*，？可以代表任意一个字符，\*可以代表任意个字符（包括0个），例如\*elc，查询elc结尾的数据

2、正则

如果要在query的内容中使用正则表达式，在两端加上正斜杠/即可。比如name:/ob[am]{2}a/

也可以使用通配符进行匹配

GET /\_search

{

"query": {

"regexp": {

"user": {

"value": "k.\*y",

"flags" : "ALL",

"max\_determinized\_states": 10000,

"rewrite": "constant\_score"

}

}

}

}

Field：user

Flags：ALL、COMPLEMENT、INTERVAL、INTERSECTION、ANYSTRING

ALL:默认值，允许所有操作符

COMPLEMENT：使用~进行排除操作a~bc # matches 'adc' and 'aec' but not 'abc'

INTERVAL：<>进行数字排行匹配

foo<1-100> # matches 'foo1', 'foo2' ... 'foo99', 'foo100'

foo<01-100> # matches 'foo01', 'foo02' ... 'foo99', 'foo100'

INTERSECTION：&功能相当于and

aaa.+&.+bbb # matches 'aaabbb'

ANYSTRING：@，类似于\*可以匹配所有字符串，不同的操作符可以组合使用

@&~(abc.+) # matches everything except terms beginning with 'abc'

max\_determinized\_states：Lucene将每个正则表达式转换为一个包含若干确定状态的具体查询条件，此参数限制分解出的条件个数，默认值10000

rewrite：此参数值得改变会影响搜索性能与相关性，精通es之前不要改动此参数，es推荐使用constant\_score, constant\_score\_boolean, top\_terms\_boost\_N

constant\_score：默认值

constant\_score\_boolean：将原始查询改为bool查询

scoring\_boolean：

top\_terms\_blended\_freqs\_N：

top\_terms\_boost\_N：

top\_terms\_N：

3、模糊查询

在query的搜索值可以使用~来进行模糊查询如“query”:”name:obalv~”

在~后面加数字来指定允许有几个字母错误，如“query”:”name:obalv~1”只能允许一个字母错误

### 11、range范围查询

GET /\_search

{

"query": {

"range" : {

"age" : {

"gte" : 10,

"lte" : 20,

"boost" : 2.0

}

}

}

}

Gt：大于

Gte：大于等于

Lt：小于

Lte：小于等于

Format：用于解析时间值

Relation：

Time\_zone:

Boost:

时间参数释义：

+1h: Add one hour

-1d: Subtract one day

/d: Round down to the nearest day

Assuming now is 2001-01-01 12:00:00, some examples are:

now+1h: now in milliseconds plus one hour. Resolves to: 2001-01-01 13:00:00

now-1h: now in milliseconds minus one hour. Resolves to: 2001-01-01 11:00:00

now-1h/d: now in milliseconds minus one hour, rounded down to UTC 00:00. Resolves to: 2001-01-01 00:00:00

2001.02.01\|\|+1M/d: 2001-02-01 in milliseconds plus one month. Resolves to: 2001-03-01 00:00:00

gt

Rounds up to the first millisecond not covered by the rounded date.

For example, 2014-11-18||/M rounds up to 2014-12-01T00:00:00.000, excluding the entire month of November.

gte

Rounds down to the first millisecond.

For example, 2014-11-18||/M rounds down to 2014-11-01T00:00:00.000, including the entire month.

lt

Rounds down to the last millisecond before the rounded value.

For example, 2014-11-18||/M rounds down to 2014-10-31T23:59:59.999, excluding the entire month of November.

lte

Rounds up to the latest millisecond in the rounding interval.

For example, 2014-11-18||/M rounds up to 2014-11-30T23:59:59.999, including the entire month.

### 12、同义词

更新同义词词库、ik分词器main.dic词库都需要重建索引，ik分词器的更改需要先重启

在"analysis-ik/synonyms.txt"路径下新建同义词文档

PUT /test11

{

"settings": {

"number\_of\_replicas": 1,

"number\_of\_shards": 1,

"analysis": {//指定分析器

"filter": {

"my\_synonym\_filter": {//自定义同义词分析器

"type": "synonym",

"synonyms\_path": "analysis-ik/synonyms.txt"

}

},

"analyzer": {//分析器

"ik\_syno": {//用于搜索

"type": "custom",

"tokenizer": "ik\_smart",

"filter": [

"my\_synonym\_filter"

]

},

"ik\_syno\_max": {//用于分词

"type": "custom",

"tokenizer": "ik\_max\_word",

"filter": [

"my\_synonym\_filter"

]

}

}

}

},

"mappings": {

"\_doc": {

"properties": {

"name": {

"type": "text",

"analyzer": "ik\_syno\_max",

"search\_analyzer": "ik\_syno"

}

}

}

}

}

PUT /test11/\_doc/1

{

"name":"苹果"

}

GET /test11/\_analyze

{

"field": "name",

"text": "气质好"

}

GET /test11/\_search

{

"query": {

"match": {

"name": "apple"

}

}

}

### 13、聚合查询

聚合查询：

**指标聚合**：对⼀个数据集求最⼤、最⼩、和、平均值等指标的聚合，在ES中称为指标聚合

**桶聚合**：⽽关系型数据库中除了有聚合函数外，还可以对查询出的数据进⾏分组group by，再在组上进⾏指标聚合。在ES中称为桶聚合

Bucket，分桶类型，类似于group by，按照一定的规则将文档分配到不同的桶中，以达到分类分析的目的

Terms根据term分桶，如果是text类型，则按照分词后的结果分桶

range（指定数值范围来设置分桶）

data range（指定日期的范围）

histogram（以固定间隔的策略分割）

date histogram（针对日期)

Metric，指标分析类型，如最大值、最小值、平均值等等

单值分析 min、max、avg、sum、cardinality(distinct count)

多值分析 stats,extended stats percentile（百分位数） percentile rank top hits

Pipeline,管道分析类型，基于上一级的聚合分析结果

Parent结果内嵌到现有的聚合分析结果

Sibling结果与现有聚合分析结果同级

Matrix,矩阵分析类型

语法

aggs->聚合名字->聚合类型->聚合内容

#### 指标聚合：

min、max、avg、sum、cardinality(distinct count 去重)value\_count 统计非空字段的文档数

stats统计count max min avg sum 5个值

Extended stats ⽐stats多4个统计结果： 平⽅和、⽅差、标准差、平均值加/减两个标准差的区间

Percentiles 占比百分位对应的值统计，默认返回（1,5,25,50,75,95,99）分位上的最大值，可以了解百分位数的概念及运算，大意是将数据从小到大排列，1%即前1%的数

size 查询结果，查询的记录条数为0，不返回记录详情，也就是，只返回聚合结果

POST /nba/\_search

{

"query": {

"term": {

"teamNameEn": {

"value": "Rockets"

}

}

},

"aggs": {

"avgAge": {

"avg": {

"field": "age"

}

}

},

"size": 0

}

百分位数

POST /nba/\_search

{

"query": {

"term": {

"teamNameEn": {

"value": "Rockets"

}

}

},

"aggs": {

"pecentAge": {

"percentiles": {

"field": "age"

}

}

},

"size": 0

}

指定具体的百分位数

POST /nba/\_search

{

"query": {

"term": {

"teamNameEn": {

"value": "Rockets"

}

}

},

"aggs": {

"percentAge": {

"percentiles": {

"field": "age",

"percents": [

20,

50,

75

]

}

}

},

"size": 0

}

#### 桶聚合：类似于MySQL的group by

1、terms Aggregation 根据字段项分组聚合

Group by age

POST /nba/\_search

{

"query": {

"term": {

"teamNameEn": {

"value": "Rockets"

}

}

},

"aggs": {

"aggsAge": {

"terms": {

"field": "age",

"size": 10

}

}

},

"size": 0

}

⽕箭队根据年龄进⾏分组，分组信息通过年龄从⼤到⼩排序 ，组内升序，组之间根据组内最大年龄降序

POST /nba/\_search

{

"query": {

"term": {

"teamNameEn": {

"value": "Rockets"

}

}

},

"aggs": {

"aggsAge": {

"terms": {

"field": "age",

"size": 10,

"order": {

"\_key": "desc"

}

}

}

},

"size": 0

}

⽕箭队根据年龄进⾏分组，分组信息通过⽂档数从⼤到⼩排序 (通过⽂档数)

POST /nba/\_search

{

"query": {

"term": {

"teamNameEn": {

"value": "Rockets"

}

}

},

"aggs": {

"aggsAge": {

"terms": {

"field": "age",

"size": 10,

"order": {

"\_count": "desc"

}

}

}

},

"size": 0

}

嵌套聚合每⽀球队按该队所有球员的平均年龄进⾏分组排序 (通过分组指标值)

POST /nba/\_search

{

"aggs": {

"aggsTeamName": {

"terms": {

"field": "teamNameEn",

"size": 30,

"order": {

"avgAge": "desc"

}

},

"aggs": {

"avgAge": {

"avg": {

"field": "age"

}

}

}

}

},

"size": 0

}

湖⼈和⽕箭队按球队平均年龄进⾏分组排序 (指定值列表)

POST /nba/\_search

{

"aggs": {

"aggsTeamName": {

"terms": {

"field": "teamNameEn",

"include": [

"Lakers",

"Rockets",

"Warriors"

],

"exclude": [

"Warriors"

],

"size": 30,

"order": {

"avgAge": "desc"

}

},

"aggs": {

"avgAge": {

"avg": {

"field": "age"

}

}

}

}

},

"size": 0

}

湖⼈和⽕箭队按球队平均年龄进⾏分组排序 (正则表达式匹配值)

POST /nba/\_search

{

"aggs": {

"aggsTeamName": {

"terms": {

"field": "teamNameEn",

"include": "Lakers|Ro.\*|Warriors.\*",

"exclude": "Warriors",

"size": 30,

"order": {

"avgAge": "desc"

}

},

"aggs": {

"avgAge": {

"avg": {

"field": "age"

}

}

}

}

},

"size": 0

}

2、 Range Aggregation范围分组聚合

NBA球员年龄按20,20-35,35这样分组

POST /nba/\_search

{

"aggs": {

"ageRange": {

"range": {

"field": "age",

"ranges": [

{

"to": 20

},

{

"from": 20,

"to": 35

},

{

"from": 35

}

]

}

}

},

"size": 0

}

NBA球员年龄按20,20-35,35这样分组 (起别名)

POST /nba/\_search

{

"aggs": {

"ageRange": {

"range": {

"field": "age",

"ranges": [

{

"to": 20,

"key": "A"

},

{

"from": 20,

"to": 35,

"key": "B"

},

{

"from": 35,

"key": "C"

}

]

}

}

},

"size": 0

}

3、Date range Aggregation 时间范围分组聚合

NBA球员按出⽣年⽉分组

POST /nba/\_search

{

"aggs": {

"birthDayRange": {

"date\_range": {

"field": "birthDay",

"format": "MM-yyy",

"ranges": [

{

"to": "01-1989"

},

{

"from": "01-1989",

"to": "01-1999"

},

{

"from": "01-1999",

"to": "01-2009"

},

{

"from": "01-2009"

}

]

}

}

},

"size": 0

}

4、Date Histogram Aggregation 时间柱状图聚合

按天、月、年进行聚合统计。可按year，quarter季度，month，day，hour，minute，second间隔聚合

POST /nba/\_search

{

"aggs": {

"birthday\_aggs": {

"date\_histogram": {

"field": "birthDay",

"format": "yyyy",

"interval": "year"

}

}

},

"size": 0

}

排序加聚合查询

GET /test/\_search

{

"query": {

"match": {

"name": "kevin"

}

},

"sort": [

{

"age": {

"order": "desc"

}

}

],

"aggs":{

"group\_by\_age":{//自定义的名称

"terms": {//统计

"field": "age",

"size": 4 分组后显示的分组个数

}

}

}

}

## 2.3 数据迁移

### 3.1 索引级别的数据迁移

ES提供了\_reindex这个API。相对于我们重新导入数据肯定会快不少，实测速度大概是bulk导入数据的5-10倍。

首先创建新的索引，然后通过如下命令将旧索引上的数据导入到新索引里

POST \_reindex

{

"source": {

"index": "old\_index"

},

"dest": {

"index": "new\_index"

}

}

数据迁移之后，通过配置中心将查询索引切换为新的索引，或者使用另一种方式，首先删除旧的索引，然后给新的索引配置别名，别名即为旧的索引

PUT /db\_index\_2/\_alias/db\_index

如果新的index中有数据，并且可能发生冲突，那么可以设置version\_type"version\_type": "internal"或者不设置，则Elasticsearch强制性的将文档转储到目标中，覆盖具有相同类型和ID的任何内容：

POST \_reindex

{

"source": {

"index": "old\_index"

},

"dest": {

"index": "new\_index",

"version\_type": "internal"

}

}

### 数据迁移效率

问题发现：

常规的如果我们只是进行少量的数据迁移利用普通的reindex就可以很好的达到要求，但是当我们发现我们需要迁移的数据量过大时，我们会发现reindex的速度会变得很慢数据量几十个G的场景下，elasticsearch reindex速度太慢，从旧索引导数据到新索引，当前最佳方案是什么？

原因分析：

reindex的核心做跨索引、跨集群的数据迁移。  
慢的原因及优化思路无非包括：  
    1）批量大小值可能太小。需要结合堆内存、线程池调整大小；  
    2）reindex的底层是scroll实现，借助scroll并行优化方式，提升效率；  
    3）跨索引、跨集群的核心是写入数据，考虑写入优化角度提升效率。

可行方案：

1）提升批量写入大小值

默认情况下，\_reindex使用1000进行批量操作，您可以在source中调整batch\_size。

批量大小设置的依据：

1、使用批量索引请求以获得最佳性能。

批量大小取决于数据、分析和集群配置，但一个好的起点是每批处理5-15 MB。

注意，这是物理大小。文档数量不是度量批量大小的好指标。例如，如果每批索引1000个文档:

1）每个1kb的1000个文档是1mb。

2）每个100kb的1000个文档是100 MB。

这些是完全不同的体积大小。

2、逐步递增文档容量大小的方式调优。

1）从大约5-15 MB的大容量开始，慢慢增加，直到你看不到性能的提升。然后开始增加批量写入的并发性(多线程等等)。

2）使用kibana、cerebro或iostat、top和ps等工具监视节点，以查看资源何时开始出现瓶颈。如果您开始接收EsRejectedExecutionException，您的集群就不能再跟上了:至少有一个资源达到了容量。要么减少并发性，或者提供更多有限的资源(例如从机械硬盘切换到ssd固态硬盘)，要么添加更多节点。

2）借助scroll的sliced提升写入效率

Reindex支持Sliced Scroll以并行化重建索引过程。这种并行化可以提高效率，并提供一种方便的方法将请求分解为更小的部分。

sliced原理（from medcl）

1）用过Scroll接口吧，很慢？如果你数据量很大，用Scroll遍历数据那确实是接受不了，现在Scroll接口可以并发来进行数据遍历了。  
2）每个Scroll请求，可以分成多个Slice请求，可以理解为切片，各Slice独立并行，利用Scroll重建或者遍历要快很多倍。

slicing使用举例

slicing的设定分为两种方式：手动设置分片、自动设置分片。  
手动设置分片参见官网。  
自动设置分片如下：

POST \_reindex?slices=5&refresh

{

"source": {

"index": "twitter"

},

"dest": {

"index": "new\_twitter"

}

}

slices大小设置注意事项：

1）slices大小的设置可以手动指定，或者设置slices设置为auto，auto的含义是：针对单索引，slices大小=分片数；针对多索引，slices=分片的最小值。  
2）当slices的数量等于索引中的分片数量时，查询性能最高效。slices大小大于分片数，非但不会提升效率，反而会增加开销。  
3）如果这个slices数字很大(例如500)，建议选择一个较低的数字，因为过大的slices 会影响性能。

# 3、lucene

Lucene与搜索引擎的关系：

Lucene是检索系统的核心框架，它类似spring Jar包。只要把Jar引入自己的工程就可以用，不依赖es 和 solr，单机版 -> 集群版。

特别适用于小规模的检索，比如10万左右的数据，用mysql达不到效果，用es和solr等成本太高，就可以直接用Lucene。直接嵌入到你的工程，不需要部署任何服务。

Mysql like ‘%四川省%’

Solr: 他们是基于Lucene封装的一个成熟的产品，提供很多的api尤其是提供了分布式索引，大大的提高了存储能力和搜索效率。适用于千万级以上分单机版和集群版：最大的区别

zookeeper:协调 管理服务 分布式锁 节点 统一动态配置的功能。分布式

Es:跟solr一样他也是基于Lucene开发的一款分布式搜索引擎。分单机版和集群版 这个集群版不需要zk;

## 3.1 trie树

Es中用于获取分词存不存在

1.什么是Trie树：trie树就是我们平常说的字典树又称前缀树，它是一种专门用来处理字符串匹配的数据结构。特别适合用来在很多字符串中快速查找某一个特定的字符串。前缀树，赫夫曼树，前缀编码

2.Trie的数据结构：假设我们有以下几个英文单词：my name apple age sex，假如我们要查找里面某一个字符串是否存在，你怎么去找呢？

散列表：hashMap

如果利用Trie树我们该怎么来解决上面的查找问题呢？

我们可以将上面的字符串变成以下结构，利用字符串的公共前缀，将重复的合在一起组成一颗树，即为我们所要讲的trie树。

3.Trie树的构建：我们要先将词分成一个个的字母，然后再依次插入到树中。如右图所示，根节点root，如果我们要插入app

则首先将app分成：a,p,p，然后从root点开始，一层层的插入，注意的是

P会挂在a下面，后面的一个p会挂在前面的p上。单词的末尾我们就用紫色表示。

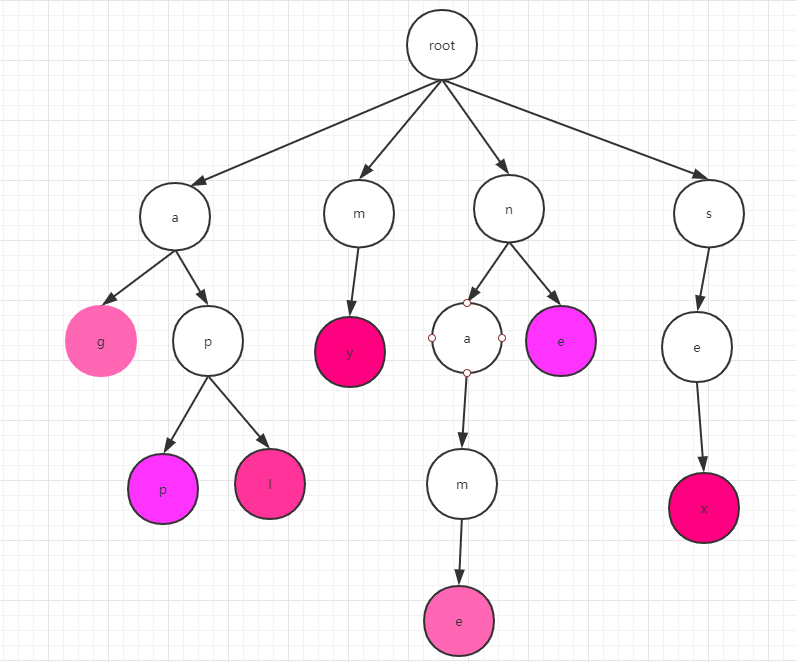
这里需要注意我们插入的时候每一层的字母都是有序的。

4.Trie的查找：

查找我们就从root点开始，再第一层找第一个字母，依次往下找到我们所要的单词。

注意要找到末尾的标记才算完成一个单词的查找。比如app，我们要找ap

虽然字典树里面有ap，但是这个p不是紫色那么ap还是不存在字典树中的。



5.Trie树的实现:其实Trie树就是一颗多叉树。这里我们应该要想到B+Tree&B-Tree，是有些类似的。

Trie树又是巧妙的利用了数组的下标，因为英文字母刚好是26个，所以我们可以开一个26长度的数组

A[] = new int[26];

A[0] = ‘a’ => 下标就是’a’-97 这里刚好就是0，利用的是ascii计算。

所以它的数据结构应该是

Class TrieNode{

Char c; //存储当前这个字符

TrieNode child[26]; //存储这个字符的子节点

}

6.Trie树的分析：

时间复杂度：非常高效 O(单词的长度)

空间复杂度：

以空间来换效率的数据结构。因为每个单词理论上都有26个子节点，所有它的空间复杂度就是26^n，n表示的是树的高度。

优化：

1.重复的字母不要重复建

2.因为我们每个node都开了26个空间来存储节点。但实际情况可能不需要这么多，所以这里其实我们可以考虑用散列表来实现，

这里大家可以去看IK的源码，当子节点少的时候是用的数组，但是节点大于3个是它是用的hashMap，这个再一定的程度上是可以节省很多的空间的。





# 4、es优化

## 4.1 系统方面优化

### 1.1 filestystem cache优化

es 的搜索引擎严重依赖于底层的 filesystem cache，你如果给 filesystem cache 更多的内存，尽量让内存可以容纳所有的 idx segment file 索引数据文件，那么你搜索的时候就基本都是走内存的，性能会非常高。所以es节点所在的机器最好将一半的内存分配给es，使其能加载更多的索引文件，提高查询性能。正常来讲如果数据量过于庞大则es这种只应该存储索引字段，从es中拿到索引字段的值，然后再去Hbase、mongodb、Hive等数据存储平台获取数据

磁盘方面配置SSD，机械盘做阵列RAID5 RAID10虽然看上去很快，但是随机IO还是SSD好

**数据预热：**

举个例子，拿微博来说，你可以把一些大V，平时看的人很多的数据，你自己提前后台搞个系统，每隔一会儿，自己的后台系统去搜索一下热数据，刷到 filesystem cache 里去，后面用户实际上来看这个热数据的时候，他们就是直接从内存里搜索了，很快。

或者是电商，你可以将平时查看最多的一些商品，比如说 iphone 8，热数据提前后台搞个程序，每隔 1 分钟自己主动访问一次，刷到 filesystem cache 里去。

对于那些你觉得比较热的，经常会有人访问的数据，最好做一个专门的缓存预热子系统，就是对热数据每隔一段时间，就提前访问一下，让数据进入 filesystem cache 里面去。这样下次别人访问的时候，一定性能会好一些

## 4.2 程序设计

### 2.1 分页优化

不允许深度分页，否则会有大量数据进入内存进行排序，将各个分片的数据放入协调点排序后返回客户端

### 4.3 mapping设计

尽量使用keyword替代一些long或者int之类的，term查询总比range查询好

评分消耗资源，如果不需要可使用filter过滤来达到关闭评分功能，score则为0，如果使用constantScoreQuery则score为1

使用自动生成的id（auto-generated ids）

索引具有显式id的文档时，Elasticsearch需要检查具有相同id的文档是否已经存在于相同的分片中，这是昂贵的操作，并且随着索引增长而变得更加昂贵。 通过使用自动生成的ID，Elasticsearch可以跳过这个检查，这使索引更快。可以使用显示id，性能下降没有那么明显