# ES安装

## window环境

### 单机单服务

在window环境下，直接将下载好的ES压缩包解压，找到解压目录下的

..\elasticsearch-5.6.9\bin下面的elasticsearch.bat，双击即可运行

打开浏览器，输入：<http://localhost:9200>，返回如下信息说明执行成功：



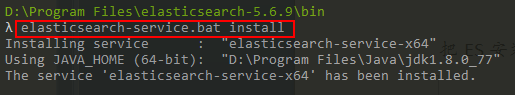
把ES安装成一个服务：到ES安装目录的bin目录，执行命令：

elasticsearch-service.bat install

以后启动ES，可以执行命令：

elasticsearch-service.bat start

也可以将该服务配置成自动启动，随开机而启动



### 单机多服务

ES在安装成服务之前，可以通过设置window环境变量，配置服务的属性：

( 注意：可能不同版本之间这些环境变量的变量名会不一样 )，正常情况，仅仅使用到SERVICE\_DISPLAY\_NAME改变ES服务名称而已：





①先设置ES服务名称：

( 我是按照ES+版本号 来安装，这个可以随意取名 )



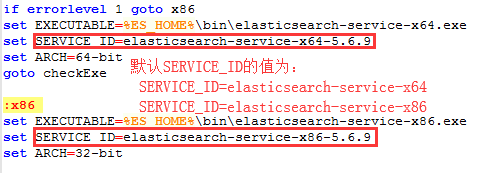
②修改ES安装服务的脚本文件elasticsearch-service.bat的SERVICE\_ID

(ES 2.x版本安装服务的脚本文件：service.bat)

在默认SERVICE\_ID值后面加上ES的版本号即可

( 当然也可以加上其他，只要保证唯一性就行，避免本机服务名冲突 )

我修改的情况如下：



③全部设置好，执行elasticsearch-service.bat install就可以安装了。

一个版本的ES安装好后，重新执行①②步，安装另一个版本的ES，结果：



## linux环境

### 安装步骤

①准备好ES的安装包，使用rz上传到Linux服务器上，或者直接联网下载，

网址为：[ES-6.4.3.tar.gz](curl%20-L%20-O%20https:/artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.4.3.tar.gz)。



②解压刚才上传或下载的ES压缩包，切换到解压后的目录下的bin目录，执行

命令：./elasticsearch（前台启动），./ elasticsearch -d（后台启动）

### 安装报错

#### 拒绝root用户启动

es不允许以root用户启动，在2.x版本还可以在启动时加上参数，如：

./elasticsearch -Des.insecure.allow.root=true来以root用户启动；

但是在5.x以后，除非修改源代码，否则不能以root用户启动，因此需要创建一个单独的用户用来运行ES。

**步骤：**

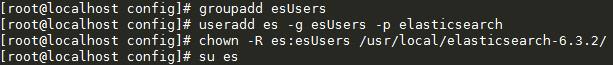
①创建用户组，执行命令：groupadd esUsers（esUsers为自定义组名）

②创建用户，执行命令：useradd es -g esUsers -p elasticsearch

( es为用户名，esUsers为用户组名，-p后面是密码 )

③更改elasticsearch文件夹及内部文件的所属用户及组，执行命令：

chown -R es:esUsers /usr/local/... (最后一个参数是ES安装目录)



#### 启动报错

①**报错原因**：“max file descriptors [4096] for elasticsearch process is too low, increase to at least [65536]”。

**解决方法：**切换到root用户，在vim /etc/security/limits.conf中添加：

\* soft nofile 65536

\* hard nofile 131072

\* soft nproc 2048

\* hard nproc 4096

②**报错原因**：“max virtual memory areas vm.max\_map\_count [65530] is too low, increase to at least [262144]”

**解决方法：**切换到root用户，修改配置vim /etc/sysctl.conf，添加如下的

配置：vm.max\_map\_count=655360，并执行命令：sysctl -p

### 启动关闭ES

**启动ES（**先切换到ES安装目录下的bin下**）**

①前台启动ES，执行：./elasticsearch

②后台启动ES，执行：./elasticsearch -d

**关闭ES**

①查找ES的进程号，执行命令：ps -ef | grep elastic

C:\Users\user\Desktop\无标题.png(2806是ps命令的端口号，不要混了)

②杀掉ES进程，执行命令：kill -9 2971 （2971是上条命令查的ES进程号）

## 导入数据

### JSON导入

准备好JSON文件，注意格式要符合\_bulk的要求，然后执行下面的语句：

curl -H "Content-Type: application/x-ndjson" -XPOST "127.0.0.1:9200/bank/account/\_bulk?pretty" --data-binary @accounts.json

# ES索引与映射

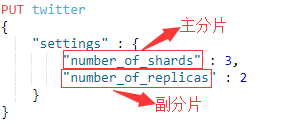
## 设置index

创建索引的时候，我们可以为索引指定一些配置，例如：

**①设置主分片数和副分片数**

默认情况下，ES索引的主副分片数为：primary shard=5，replica shard=1

(5个主分片，每个主分片对应1个副分片)。使用 setting API



上面的参数意思是：创建3个主分片，每个主分片附带2个副分片，所以总共

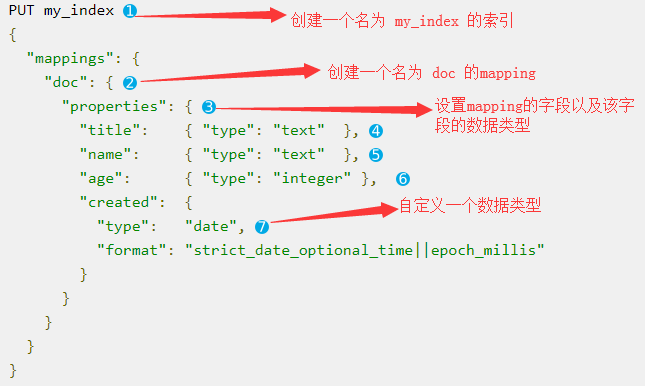
有3+3\*2=9个分片

## 设置mapping

### 自定义映射

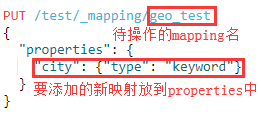
虽然ES可以根据我们添加的数据自动映射，但是很多情况下还是需要我们手动

配置mapping，使用下面的语法：



### 添加新映射

①mapping创建后可以添加新的字段，但是不能删除或修改：



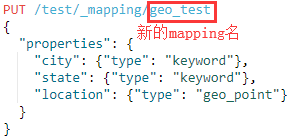
②在创建index的时候，直接设置多个mapping；



如果创建索引后，需要添加新的映射，可以继续为它添加新的mapping。

**注意：**在ES6.x版本后一个index只能创建一个type，而在ES7.x版本后，

type被完全废除掉：



### 设置分词器

当自定义映射为字符串类型时，可以指定它的分词器



### 查看映射

查看索引的映射可以使用\_mapping API，例如：

①查看索引所有映射：GET /test #查看test索引的所有映射；

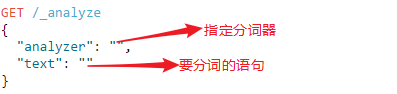
②查看索引单个映射：GET /test/\_mapping/test1 #查看test1映射

# ES分词

## \_analyze

可以使用\_analyze API，查看分词器对语句的分词效果；它有两种用法：

①指定分词器对语句直接进行分词，语法：



②指定文档的字段对语句进行分词，语法：





## 分词器

### 标准分词器

standard analyzer，即标准分词器。它是ES默认的分词器，如果未指定，则使用该分析器。它提供基于语法的标记化（基于Unicode文本分割算法），适用于大多数语言：



注：

ES的标准分词器对中文很不友好，都是单字分词；

### IK分词器

IKAnalyzer是一个开源的，基于java语言开发的轻量级的中文分词工具包。

IK Analysis插件将IKAnalyzer集成到elasticsearch中，支持自定义词典，已经托管到github上：

<https://github.com/medcl/elasticsearch-analysis-ik>

#### 安装IK Analysis

**ES安装IK Analysis方式1：**

①下载IK的编译包：elasticsearch-analysis-ik-5.6.9.zip

②在ES安装目录的plugins目录下，创建ik目录，将编译包解压到ik目录。

**注意：**ES的安装路径不能出现空格，否则ik插件安装进去会导致整个ES数

据目录报错，ES虽然可以启动，但是kibana无法访问到ES.

**ES安装IK Analysis方式2：**

①切换到ES安装目录下的bin目录，执行命令：

./bin/elasticsearch-plugin install https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v6.3.0/elasticsearch-analysis-ik-6.3.0.zip

其中 6.3.0需要替换成我们自己ES的版本号。

②重启ES即可生效。(注意ES安装路径也不要出现空格)

#### 使用IK分词器

IK Analysis插件的分词器名称为：ik\_smart和ik\_max\_word

#### 配置IK分词器

在IK分词器插件的config目录下，有个IKAnalyzer.cfg.xml文件，可以通过它指定我们自己的词典：



# 文档处理API

## 新增文档

### 基本语法

ES新增文档，使用的HTTP请求方式为"put"和"post"。其中，put方式可以自定义\_id，post方式是让ES自生成\_id。

**语法：**

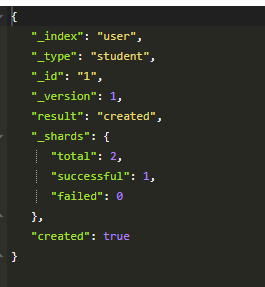
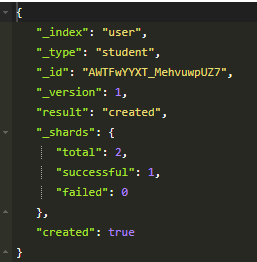
put <index>/<type>/<\_id>

post <index>/<type>

**例子：**

ES中每个文档都有版本号( 下图中的\_version)，每当文档变化（修改、删除）都会使 \_version 增加，如果添加文档成功，返回如下信息：

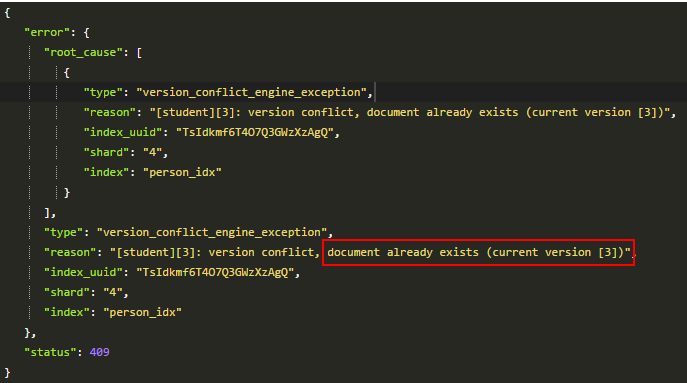
### 唯一文档

在添加文档时，想让ES在文档不存在时才添加，只要保证索引、类型、ID三者不同，就可以保证文档为新添加而不是修改的。使用POST方式添加的文档肯定是新文档，因为ID由ES自动添加，但是如果ID想自己添加，就需要使用PUT方式，可以使用如下2种语法：

①PUT /\_index/\_type/\_id?op\_type=create

②PUT /\_index/\_type/\_id/\_create

如果原文档不存在则新添加，如果原文档存在，则报错：



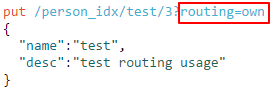
### 自定义路由

ES默认会将\_id取hash值后对primary\_shards求余的值来决定将这个文档存到那个shards上：



如果业务逻辑上有关联的文档，需要将它们存到同一个shard上，但由于它们的\_id有大区别，导致分片算法后ES会将它们保存到不同的shard上。这时候，使用routing可以形成个性化文档分片映射。个性化的routing值可以确保相关的文档存储到同样的分片上。

PS：创建文档时用了routing，查询此文档的时候也要用routing：

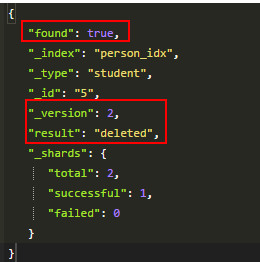
## 删除文档

### 基本语法

当使用delete方式的HTTP请求时，可以删除一个文档，语法为：

DELETE \_index/\_type/\_id。例如： 

如果删除成功，返回found:true，result:deleted，且版本号被增加了：



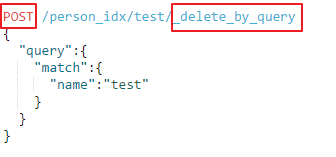
删除一个文档不会立即从磁盘上移除，它只是被标记成已删除。Elasticsearch 将会在之后添加更多索引时才会在后台进行删除内容的清理

### 查询删除

\_delete\_by\_query API可以通过查询匹配到文档后删除，如何查询匹配需要查看[DSL语法](#_DSL语法)。注意：使用 \_delete\_by\_query ，HTTP请求方式是POST不是DELETE。

**例子：**

删除name字段为test的文档



### 指定路由删除

同样地，如果文档在创建时[自定义路由](#_自定义路由)routing，在删除的时候同样也需要指定routing： 

## 修改文档

ES修改文档其实分为"**全局修改**"和"**局部修改**"

### 全部修改

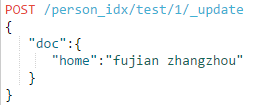
全局修改文档，即用一个全新的文档替换到旧文档，方式很简单，直接使用put方式发起相同Index-type-id的请求，当执行PUT请求时，ES发现同\_index，同\_type下已经存在同\_id的文档，它会标记旧文档为删除状态，添加新文档并重新索引。ES不会立即删除旧文档但也不能访问，它在后台等待时机成熟时默默地清除...全局修改返回的信息如下：



### 局部修改

局部修改使用的API为：\_update和doc，支持基于脚本修改文档，使用的HTTP请求方式是POST，语法为：POST <\_index>/<\_type>/<\_id>/\_update。

当文档字段不存在时，可以为文档添加新字段；当文档字段存在时，可以修改文档字段的内容：



虽然是局部修改，但ES内部执行的过程如下：

1、从旧文档中检索待修改的document

2、执行修改

3、删除旧文档(逻辑删除)

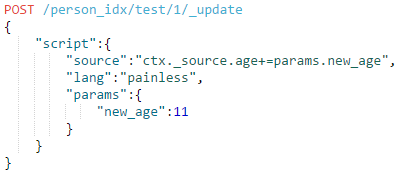
4、索引新文档

### 脚本修改

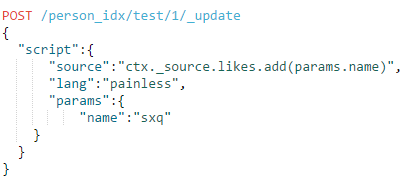
使用脚本来修改文档的内容，脚本语言用的是“painless”，在\_update API中，使用"ctx"来表示文档，所以用"ctx.\_source"表示文档内容。

**例子：**

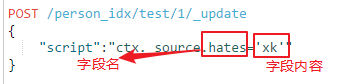
①修改整型或字符串的值，可以使用"+="，其中"params"代表参数



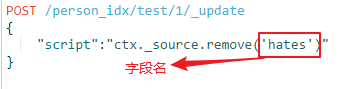
②使用add函数可以往数组中添加内容



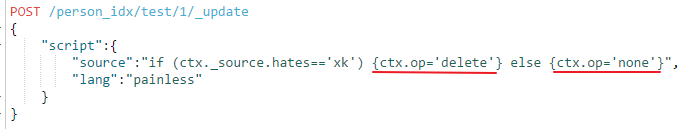
③直接向文档中添加新字段：



④使用remove()函数可以删除文档中的字段：



⑤符合某个条件时删除整个文档，使用ctx\_op

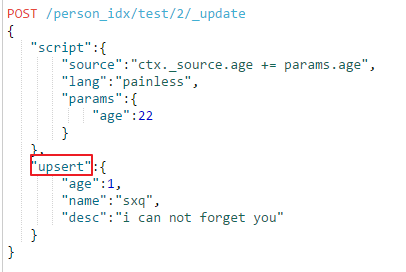


⑥如果脚本复杂，也可以单独写成一个脚本文件，放在es安装目录：

/config/scripts/，然后指定这个脚本文件即可

### upsert

在修改文档内容时，使用upsert，可以在文档存在时执行script脚本，在文档不存在时执行upsert创建文档：

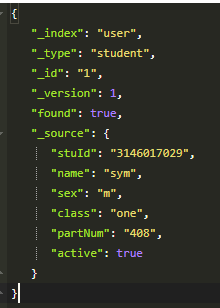
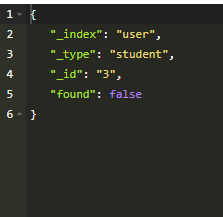


## 查询文档

### 单个查询

使用Get方式的HTTP请求，通过\_index、\_type、\_id便可以获取文档内容

语法： GET /<\_index>/<\_type>/<\_id>。若存在该文档，返回结果会是found:true，文档的数据会被放在\_source字段；若不存在该文档，返回found:false：

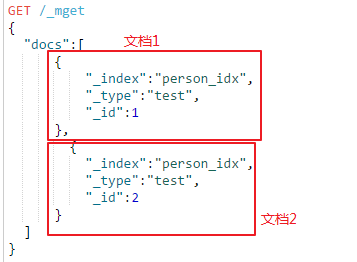
### 多个查询

多个文档查询可以使用mget或ids两个API，不过都需要同时指定\_index、\_type和\_id：

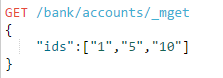
①megt通过\_index、\_type、\_id来同时获取多个文档，如果url不加索引，

则在请求体中需要指定\_index，如果url加了索引，则请求体只需要\_type

和\_id就行，同理如果url加了索引和类型，则请求体只要"\_id"：



②当只需要用\_id来获取多个文档，可以直接用ids:



### 文档存在判断

使用HEAD可以用于判断文档是否存在，如果存在返回202，不存在返回404

### 指定路由查询

当文档是以[自定义路由routing](#_自定义路由)新增到ES中，则用get API查询文档也需要使用routing：



## 批量处理

### bulk

bulk API允许在一次请求中执行多个“**索引、删除、修改**”操作，它不仅仅局限于单个索引，相反的，它可以同时操作多个索引。使用的HTTP方式是POST，

API是\_bulk。注意：每一行都需要"/n"回车(包括最后一行)，再以代码操作ES尤其需要注意，如果在工具如sense、kibana中直接回车即可。

**语法：**

POST \_bulk

{ "index" : { "\_index" : "test", "\_type" : "type1", "\_id" : "1" } }

{ "field1" : "value1" }

{ "delete" : { "\_index" : "test", "\_type" : "type1", "\_id" : "2" } }

{ "create" : { "\_index" : "test", "\_type" : "type1", "\_id" : "3" } }

{ "field1" : "value3" }

{ "update" : {"\_id" : "1", "\_type" : "type1", "\_index" : "test"} }

{ "doc" : {"field2" : "value2"} }

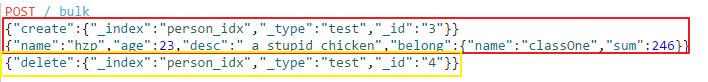
**说明：**

index/delete/create/update，指明此次操作的类型是创建、修改还是删除?

\_index、\_type、\_id，指明操作的文档。除了delete操作不需要请求体以外，其它三个都需要请求体，用来表示文档

**例子：**

红色框代表创建一个新文档、黄色框代表删除指定文档



### bulk性能分析

有没有发现为什么\_bulk API的查询格式需要将JSON按"\n"一个一个分开，其实是跟底层原理有关的：

①bulk的操作一般都是要转发不同node(节点)的shard(分片)去执行，如果采

用可读性优良的JSON数组格式，ES取到这种格式的JSON串，会做如下处理：

1、将json数组解析成JSON Array对象，此时数据就会在内存出现一份一

模一样的拷贝，一份数据是JSON文本，一份是JSON Array对象

2、解析JSON Array中的每个json，对每个请求中的document进行路由

3、为路由到同一个shard上的请求，创建一个请求数组

4、将这个数组序列化，将序列化后的请求数组发送对应的节点上去

②上述这种方式，将会耗费更多的内存，且请求一次后这些JSON Array都需要

JVM回收，引发更多的gc 开销，设想一下，如果有多个bulk请求同时发起，

每次都将JSON拷贝成数组，即成倍的内存开销与gc回收...太耗性能了

③如果使用{}\n{}\n..这种方式，ES不用将其转换成JSON Array，直接按照

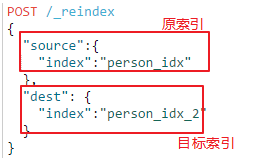
换行符切割json，对每两个一组的json，读取meta，进行document路由，

直接发送到node上去

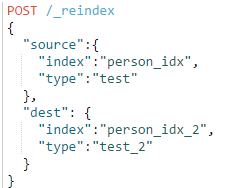
## 复制索引

### reindex

\_reindex API可以复制一个索引的内容到另一个索引上，它既不主动设置目标索引，也不会复制源索引的设置。应该在运行\_reindex操作之前设置目标索引，包括设置映射、碎片计数、副本等。使用的HTTP请求方式是POST，最简单的语法仅仅指明，原索引和目标索引:



也可以指定复制的原索引和原类型，目标索引和目标类型：



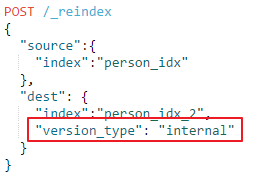
### version\_type

在"dest"中使用version\_type控制目标索引的版本号。

①version\_type=internal

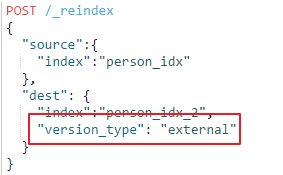
若新的index中有数据，并且可能发生冲突，那么可以设置version\_type=

internal 或 不设置，则ES强制性地将文档转储到目标中，覆盖具有相同类型和ID的任何内容



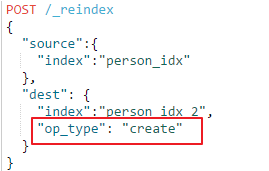
②version\_type=external

可以将原索引中的\_version复制到目标索引上保持一致



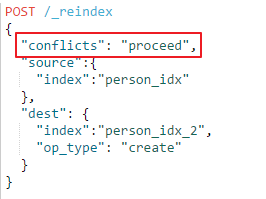
### op\_type

在"dest"中设置"op\_type"="create"可以让\_reindex只复制目标索引中没有的文档，所有已存在的文档会引发版本冲突(version conflict):



默认情况下，版本冲突会中止\_reindex的执行，可以在请求体中设置

"conflicts": "proceed"对版本冲突的文档进行计数以完成复制操作



### query

在"source"中添加"query"可以让\_reindex只复制匹配query的文档到目标索引中：



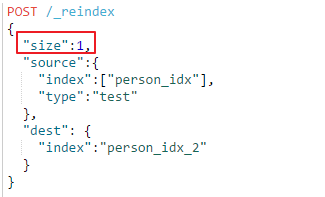
### 多个源索引

将"source"中的"index"和"type"写成数组，可以将多个源索引同时复制到目标索引中

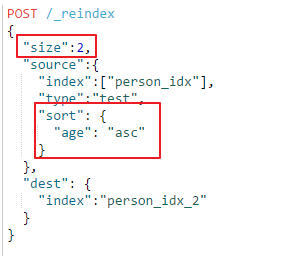


### size&sort

在请求体中使用size参数可以限制原索引拷贝到目标索引的文档个数：



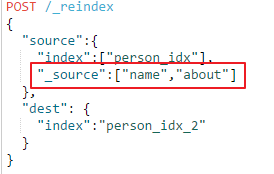
size配合sort使用，能将原索引中的文档按照一定顺序排列后拷贝指定数量到目标文档中：



### source

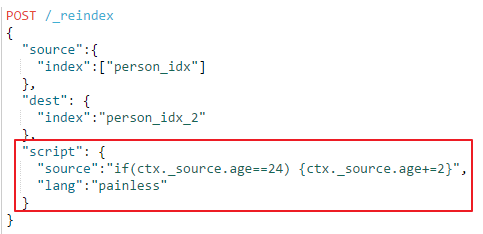
在源索引中使用\_source可以指定源索引文档的哪几个属性需要拷贝到目标索引，如果源文档中没有指定的属性，它也会被拷贝，只不过拷贝到目标索引中的

\_source字段为空对象：



### script

请求体中使用script属性，可以在拷贝过程中动态的改变源索引文档的内容



# 搜索API

搜索API允许执行搜索查询并返回与查询匹配的信息，ES支持两种搜索使用方式：①通过提供请求参数跟在URI后面来执行搜索请求，即URI的方式，例如：

②通过在请求主体内使用[查询DSL](#_DSL语法)来执行搜索请求，即request body方式，例如：



但是，大部分情况下，使用第②种请求主体的方式是最理想的。此文档以②为主。

## \_search

\_search API，最常用的搜索API，包括[DSL](#_DSL语法)和[aggs](#_ES聚合)都会使用到它，使用

\_search API搜索我们需要的数据，以请求体（request body）的方式查询，是最灵活且常用的。

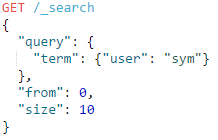
### query

query是搜索请求主体（request body）中的查询元素，允许使用[DSL语法](#_DSL语法)定义查询：



### from,size

from和size，是搜索请求主体（request body）中的分页元素，from表示从哪一条开始，默认值为0；size表示查询几条，默认值为10。例子：



**注意点：**

ES的数据都是被分配到不同的shard，在集群状态可能还分布在不同的主机上，当对数据进行分页查询时，ES是如何将数据排序后返回？假设有3个shard，每个shard上有10000条数据；一页有10条数据，当查询第一页时，每个shard上先自我排序，取出前10条，这样子就有30条数据，然后在对这30条数据排序，取出前10条。

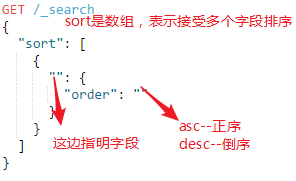
这种前几页的数据还是很迅速的，但是如果要查询第900页数据，即第9000~9009条数据，这样子每个shard同样先自我排序，取出前9099条数据，这样子就有27017条数据，然后再对27017条数据排序取第9000~9009条。

在大数据量下，这种查询超级消耗性能，称为deep\_paging，尽量避免...

### sort

sort是搜索请求主体（request body）中的排序元素。ES默认排序是按照\_score(相关性得分)降序排列，很多情况下不需要\_score排序而是需要指定字段排列，默认的字段类型为integer、long、date

...等都可以拿来排序，排序语法为：



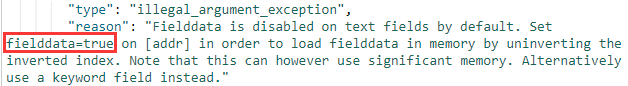
**例子：**



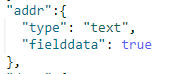
#### 字符串排序

当字段类型为字符串(ES之前版本字符串写法是"string"，现版本写法是"text")，默认是不能拿来排序的，会报错。提示需要为排序字段设置

"fielddata=true"，如下图所示



这就要求在创建索引的时候，必须手动指定mapping，语法请看[text数据类型](#_text_datatype)。在为字段创建数据类型type时，同时设置"fielddata=true"：



如果仅仅只是设置fielddata=true，ES会把排序字段的值拿去分词，提取分词来排序，例如：假设addr:"China fujian zhangzhou"，使用addr排序时，ES可能就只提取China排序，而不是将整个addr的值拿去排序。所以，如果需要整个字段的值都拿去排序，在指定字段数据类型的同时，还需要额外添加一个fields属性，它可以用来做排序字段，ES版本升级后改动较大，这里一起做个笔记：

①ES之前版本（2.x版本）写法：

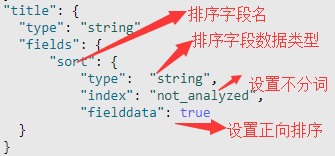
ES之前版本，表示数据类型为字符串的写法为"string"，title字段用来查

询使用，title.sort字段用来排序，并为title.sort设置fielddata=true。

这里解惑一下，当put数据时，数据一份给title，一份给sort，只不过它

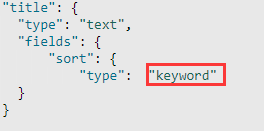
们用途不一样，对数据也处理不一样， 排序的时候，这样写即可：

"title.sort": {"order": "asc"}

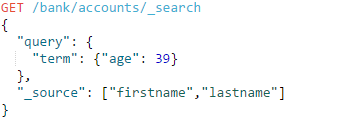
②ES现版本（5.x以上版本）写法：

ES现版本表示数据类型为字符串的写法改为"text"，整体思路还是一样的，还是创建一个title字段用于搜索，一个title.sort字段用于排序，只不过title.sort的定义变得简单多了，只要设置type=keyword。排序时，直接指定"title.sort": {"order": "asc"}即可

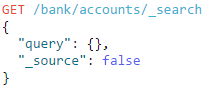
 

### \_source

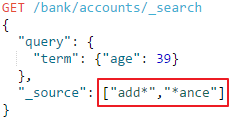
\_source是搜索请求主体（request body）中的返回字段元素。它控制每次返回\_source字段的方式，将需要返回的字段写到\_source元素内：



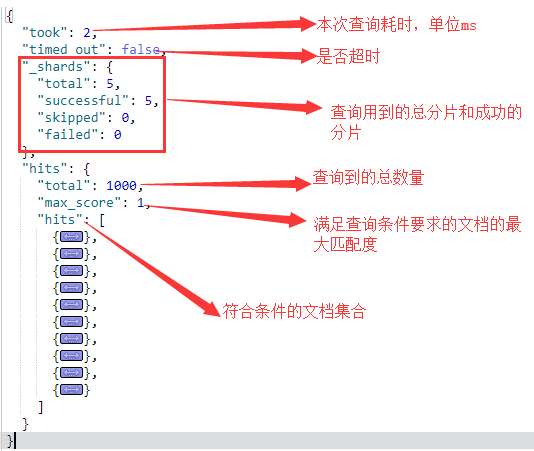
如果不想返回\_source字段，将其设置为false：



另外，\_source还可以用通配符来表示要返回的字段：



### 返回结果解析



需要强调的是：timeout机制，ES的timeout(超时)机制，并不是说超过指定时间就查询失败而报错，它的timeout机制意思是：在timeout规定的时间内返回ES能查询到的数据，不管是否查询完整，说白了，就是能查几条就是几条，

在规定的时间，你ES给我返回这几条就行...ES默认不设置超时，如果想设置超时，在查询语句加上timeout参数，如：GET /\_search?timeout=3ms

(单位要写，ms代表毫秒、s代表秒、m代表分钟...)

# DSL语法

DSL，全称是 Domain Specific Language，域特定语言。ES将Query DSL视为查询的AST（抽象语法树），由两种类型的子句组成:

①叶查询子句，查找特定字段中的特定值，例如匹配，术语或范围查询；

②复合查询子句，包装其他叶子或复合查询，用于以逻辑方式组合多个查询

## 上下文

query子句的行为取决于它是在查询上下文中还是在过滤器上下文中使用。

### query context

查询上下文中使用的查询子句回答了问题"此文档与此查询子句的匹配程度如何？"除了确定文档是否匹配之外，查询子句还计算一个\_score，表示文档相对于其他文档的匹配程度

### filter context

过滤器上下文中使用的查询子句回答问题"此文档是否与此查询子句匹配"

答案是简单的是或否，不计算任何分数。 过滤器上下文主要用于过滤结构化数据，例如：

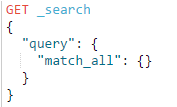
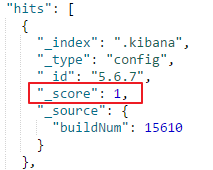
1. 文档的时间戳字段是否属于2015年至2016年的范围？

2. 文档的状态字段是否设置为“已发布”？

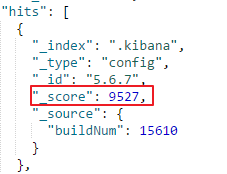
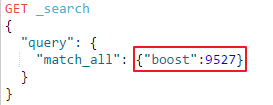
## 匹配所有

### match\_all

match\_all 是最简单的DSL查询语句，用于查询所有的文档，并且查询匹配到的文档的相关度\_score默认都为1：

如果想自定义\_score的值，可以通过给match\_all参数："boost:n"：



## 全文检索

full text query即全文检索，先建立索引，再对索引进行搜索的过程就叫全文检索。

### match

match query，即匹配查询，接受text/numerics/dates查询参数，分析它们并构造查询。匹配查询的类型为boolean，它会分析客户端提供的查询文本，分析过程根据提供的文本构造布尔查询，它的语法为：

**额外查询参数：**

①operator参数可以设置为其它控制布尔子句（默认为或）

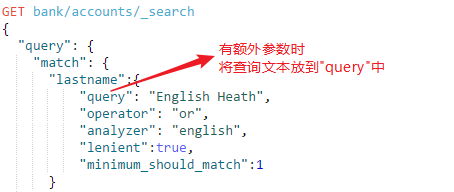
②minimum\_should\_match参数设置要匹配的可选should子句的最小数量

③analyzer来控制哪个分析仪将对文本执行分析过程。默认为字段显式映射定

义或默认搜索分析器

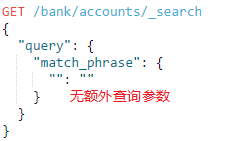
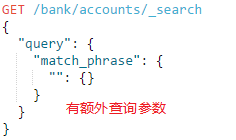
④lenient参数设置为true可以忽略由数据类型不匹配引起的异常，例如尝试

使用文本查询字符串查询数字字段默认为false。



### match\_phrase

match phrase query，即匹配短语查询，它其实是在match的基础上将operator改为and的效果，只有文档的字段完全匹配查询文本才会该文档返回。它的语法为：（语法和match很相似）

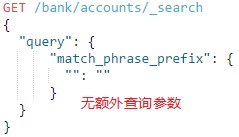
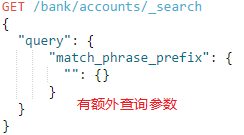
**额外查询参数：**

①analyzer：指定分析仪对查询文本进行分析查询



### match\_phrase\_prefix

match phrase prefix query，即匹配短语前缀查询，它与match\_phrase相同，只是它允许文本中最后一个术语的前缀匹配，语法为：

**额外查询参数：**

①analyzer：设置分析仪对查询文本近分析查询

②max\_expansions：设置满足查询前缀的后缀匹配数量，默认是50个。例如

有短语"ant"、"area"、"ab"、"ananlyzer"，在match\_phrase\_prefix="a"

情况下，设置max\_expansions=1，只有ab符合，设置max\_expansions=3，

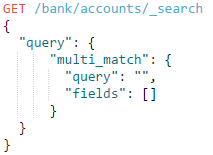
ab、ant、area都符合...以此类推，它默认是50个，官方强烈建议要将这

个设置成符合业务场景的值，避免无意义的查询



### multi\_match

multi match query，即多重匹配查询，在match的基础上允许多个字段查询，multi\_match默认带有2个属性：query 和 fields，query设置搜索文档，fields用于指定匹配字段，它的语法为：



fields可以: ①模糊匹配，如："fields": ["title", "\*\_name"]

②重要字段，如："fields" : [ "subject^3", "message" ]

(表示subject字段是message字段的三倍重要)

**额外参数：**

1、type：设置multi\_match在内部查询的执行方式，它有5个取值：

①best\_fields：默认值，它可以查询与任何字段匹配的文档，但使用最佳字段

的\_score作为返回结果的\_score(相关度)

②most\_fields：查找与任何字段匹配的文档，并组合每个字段的\_score

③cross\_fields：使用相同的分析器处理每个字段

④phrase：在每个字段上运行match\_phrase查询，并组合每个字段的\_score

⑤phrase\_prefix：在每个字段上运行match\_phrase\_prefix查询，并组合每

个字段的\_score

### common terms

common terms query，即常用术语查询，ES旧版本为了防止像"the"、

"a"等高频词影响查询性能和效果，忽略高频率的术语，通过将高频词视为一个停用词，直接将高频词过滤。常用术语查询就是替代停用词的新技术。

常用术语查询将查询术语划分为两组：更重要（即低频术语）和不太重要（即以前曾是停用词的高频术语)

它的查询步骤为：

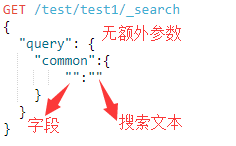
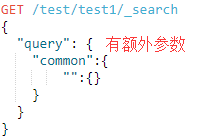
①先查询匹配低频术语(重要性大)的文档。这些术语出现在较少的文档中，对相

关性有较大影响

②基于第①步查到的结果上，对高频词(不太重要)的术语执行第二次查询，这些

术语经常出现并且对相关性的影响很小

语法：

**额外属性：**

①cutoff\_frequency：将术语分配给高频或低频组，cutoff\_frequency可以

指定为绝对频率（> = 1）或相对频率(0.0 ... 1.0)，文档频率是按照每个

分片级别计算的

②low\_freq\_operator：设置低频词的操作符(默认是 or )

③high\_freq\_operator：设置高频词的操作符(默认是 or )

④minimum\_should\_match：指定

若没有特殊指明，则表示必须存在的低频项的最小数量或百分比：

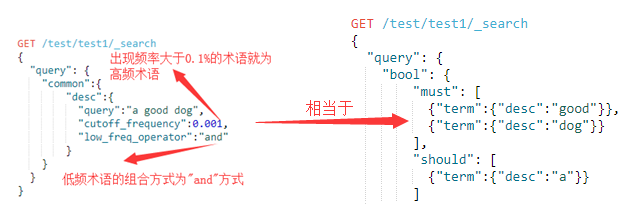
"minimum\_should\_match":2 // 低频术语占2个

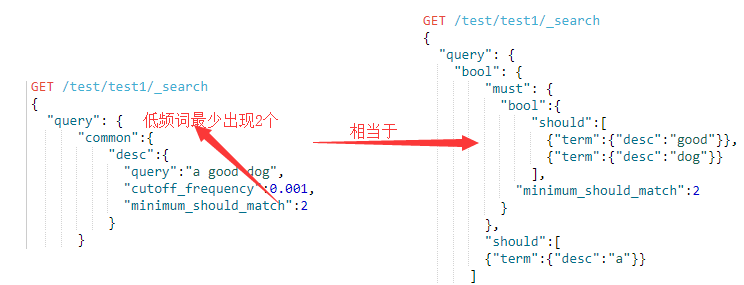
若指明为高频和低频占比，则分别表示：

"minimum\_should\_match": {

"low\_freq" : 2, // 低频术语占2个

"high\_freq" : 3 // 高频术语占1个 }





## 术语查询

term-level queries，术语级查询，将根据存储在反向索引中的确切术语进行操作，通常用于结构化数据，如数字，日期和枚举，而不是全文字段。

所有属于term level query的API的查询术语，都是给什么就认什么，不会再分析查询术语(明显区别与full text query)

### term

term query查找指定字段的倒排索引中指定的确切术语的文档。它对大小写敏感且不会对查询参数分析，所以常用在对关键字、数字、日期字段查找值。

其实，当字段的数据类型是"text"，最好不要使用term查询，为啥？ES索引"text"类型的数据，会先将它们分析，例如原句"Quick Foxes! "经过分析器后得到结果"quick"和"foxes"，最后ES是将"quick"和"foxes"保存到倒排索引。对比下使用match和term查询就可以知道：

match: "Quick Foxes!"，ES分析后得到quick和foxes去匹配倒排索引，

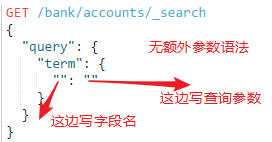
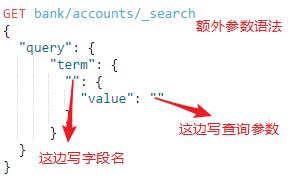
那肯定匹配的到

term: "Quick Foxes!"，ES不会对term query进行分析，所以直接用Quick

Foxes!去匹配倒排索引，肯定匹配不到(倒排索引只

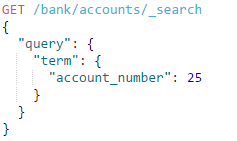
有"quick"和"foxes"，根本没有Quick Foxes!)

**语法：**

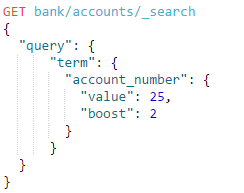
 

**例如：**

查询 account\_number=25的文档：

****

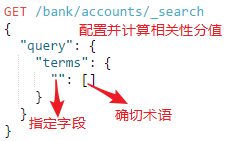
查询account\_number=25的文档，为其相关性得分设置为2：

****

### terms

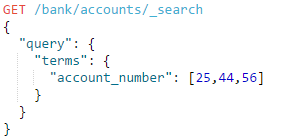
terms query用于查找包含指定字段中指定的任何确切术语的文档，它是term的"复数形式"，term只能从指定字段中的匹配一个确切术语，terms可以从指定字段中匹配多个确切术语。

**语法：**

**例如：**

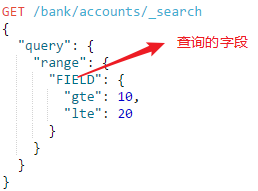
查询account\_number值为25或44或56的文档



### range

range query查找指定字段包含指定范围内的值（日期，数字或字符串）的文档。range接受以下参数：gte 、gt、lte、lt和boost(表示匹配分值)，

**语法：**



特别地，当range用在date类型上的字段时，可以使用[date Math](#_Date_Math)指定范围。

它会将日期舍入到最近的日期，月份，小时等，舍入日期取决是要"大于"还是"大于等于"，这一点很重要，不清楚的话会多出或少掉临界点的日期。

**舍入规则：**

向上舍入移动到舍入范围的最后一毫秒，并向下舍入到舍入范围的第一毫秒

①gt: 大于日期的四舍五入：2014-11-18 || / M成为2014-11-30T23：59：

59.999，即不包括整月

②gte: 大于或等于日期向下舍入：2014-11-18 || / M成为2014-11-01，即

包括整个月

③lt: 低于日期向下舍入：2014-11-18 || / M成为2014-11-01，即不包括

整个月

④lte: 小于或等于向上舍入的日期：2014-11-18 || / M成为2014-11-30T23：

59：59.999，即包括整个月

默认ES储存日期的格式为严格模式，"yyyy/MM/dd HH:mm:ss"不足两位数一定要补0，例如不能写"2018/1/11"而是要写"2018/01/11"。

参数"format"指定日期格式

由于"time\_zone"存在，所以gte指定的日期实际为"2014-12-31 23:00:00"，而"now"不受time\_zone影响

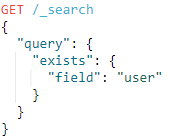
### exists

查找指定字段包含至少包含一个非空值的文档，语法：

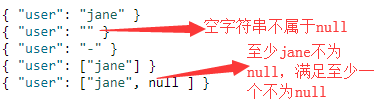


**例子：**

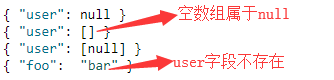
查询user字段不为空的文档：



1、会被匹配到的文档：



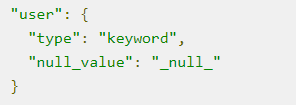
2、不会匹配到的文档：



**注意点：**

①若字段的\_mapping中包含null\_value设置，则使用指定的null\_value替

换显式空值。例如，如果user字段映射如下



显式空值(null)将被索引成字符串"\_null\_"，以下文档将匹配存在:



但是，这些文档因为没有显式空值(null)，在user字段中仍然没有值，因此与exists过滤器不匹配：



### prefix

匹配包含具有指定前缀（未分析）术语的字段的文档。前缀查询映射到

Lucene PrefixQuery，请注意"prefix"也是属于确切属于行列，与term一样，只不过term给出全部术语，而prefix只给出前缀术语：

**语法：**



**例子：**

查询address字段中带有"put"前缀的文档，注意prefix与term一样，它不会分析查询参数，"put"和"Put"是两个不同的查询参数，它也是从字段的倒排索引中匹配是否有put前缀的术语。



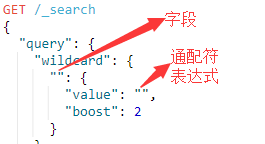
### wildcard

匹配具有与通配符表达式匹配的字段（未分析）的文档。支持的通配符是:

①"\*"，它匹配任何字符序列（包括空字符序列）②"？"，它匹配任何单个字符。

此查询可能很慢，因为它需要迭代多个术语。为了防止极慢的通配符查询，查询术语不应该以通配符\*或?之一开头。通配符查询映射到Lucene WildcardQuery。

**语法：**

**例子：**

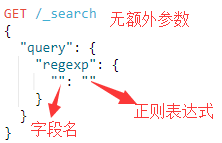
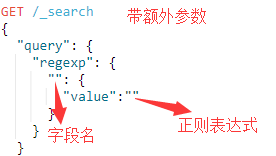
查询user字段匹配" ki\*y "通配符的文档



### regexp

查找指定字段包含与指定正则表达式匹配的术语的文档，正则表达式查询的性能在很大程度上取决于所选的正则表达式，像" .\*？+ "这样的通配符匹配器会降低性能。ES不会解释正则表达式，而只是解释支持的运算符。

**语法：**

**额外参数：**

①boost：设置匹配相关度分值

②flags：Lucecne的特殊标志，详见[Lucene API文档](http://people.apache.org/~rmuir/jacoco-core/org.apache.lucene.util.automaton/RegExp.java.html)

③max\_determinized\_states：提高此参数可以允许执行更复杂的正则表达式，

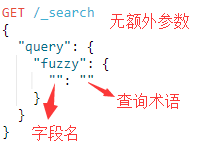
默认为10000

**例子：**

### fuzzy

fuzzy query，模糊查询，使用基于Levenshtein编辑距离的相似性。它查询所有可能的匹配项，这些匹配项处于查询参数指定的最大编辑距离内，然后检查术语词典以找出索引中实际存在哪些生成的术语。

**语法：**



带有额外查询参数的例子为：



①fuzziness：最大编辑距离。默认为AUTO

②boost：匹配相关度

③prefix\_length：不会“模糊化”的初始字符数。这有助于减少必须检查的

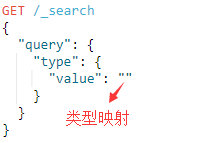
术语数量。默认为0

④max\_expansions：模糊查询将扩展到的最大术语数。默认为50

### type

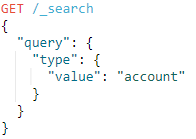
type query，查询与提供的文档/映射类型匹配的文档。

**语法：**



**例子：**

查询类型为account的文档：



### ids

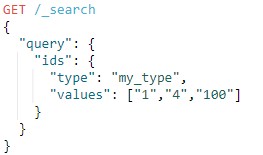
ids query，查询包含提供的ID的文档。请注意，此查询使用\_uid字段。

类型是可选的，可以省略，也可以接受值数组。如果未指定类型，则尝试在索引映射中定义的所有类型

**语法：**



**例子：**



## 复合查询

Compound queries，复合查询，包装其他复合或叶子查询，以组合其结果和分数，更改其行为，或从查询切换到筛选器上下文

### constant\_score

constant\_score query包装另一个查询(此查询必须是filter类型)，

它改变过滤器中每个文档的查询常量分数，默认为1，可以用boost参数修改。

constant\_score映射到Lucene ConstantScoreQuery。

**语法：**



**例子：**



### bool

bool query，布尔查询，将其它查询子句(如match、term...)拼凑成一个布尔组合的文档匹配查询。此查询映射到Lucene BooleanQuery，是由一个或多个布尔子句构建的，每个子句都有一个类型化的子句，它们分别是：

①must: 查询子句必须出现在匹配的文档中，并有助于相关性得分

②filter: 查询子句必须出现在匹配的文档中，但是，忽略掉相关性得分

③should: 查询子句可能出现在匹配的文档中

④must\_not: 查询子句不得出现在匹配的文档中

bool查询将匹配must或should子句的分数将被加在一起，以便为每个文档提供最终的\_score。

**请注意：**

①如果bool query是在过滤器(filter)上下文中使用并且它具有should子

句，那么至少需要一个should子句来匹配

②可以通过"minimum\_should\_match"来指定"should"子句的最小匹配度，

若minimum\_should\_match=1则文档至少需要满足"should"的一个条件，

若minimum\_should\_match=2则文档至少需要满足"should"的两个条件，

...依次类推

它出现在有"should"的bool查询中，且紧邻"should"后面书写

**语法：**

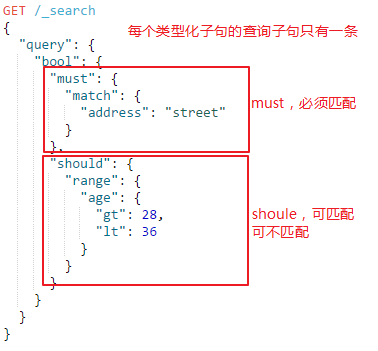
bool查询的4个类型化子句：must、must\_not、should、filter，当查询子句只有一条时，它们表示为对象，使用符号"{}"；当查询子句有多条时，它们表示为数组，使用符号"[]"，每条查询子句定义成一个对象，使用符号"{}"。

例子：

①查询address字段带有"street"的文档，age字段的取值范围在(28,36)之

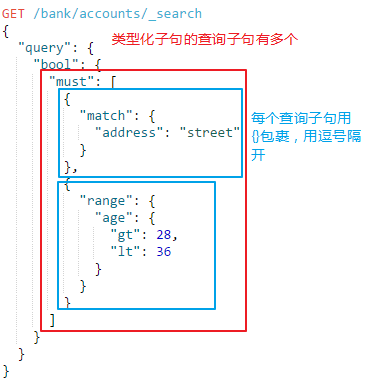
间那最好了(意味着加分)，如果不在也还行(不会加分)：



②查询address字段带有"street"，且age字段在(28,36)之间的文档，

当有多个查询子句时，类型化子句使用"[]"，类型化子句里面的查询子句需

要用"{}"包裹，查询子句之间用逗号隔开：

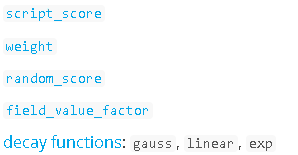


### function\_score

function\_score可以修改查询检索到的文档的\_score，使用它必须定义一个查询语句和一个或多个函数，这些函数用来计算查询匹配到的文档的新分数。当得分函数在计算上是昂贵的并且足以在过滤的文档集上计算得分，function\_

score会变得很有用。(老实说，官网的function\_score介绍我看不懂 ... [官方文档](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-function-score-query.html) ，如果真的有用到再去看官网，这边笔记只做个介绍)

function\_score提供了几种类型的得分函数：



①script\_score，允许使用脚本表达式包装另一个查询并使用从doc中的其它

数字字段值派生的计算来自定义它的评分。若使用缓存脚本编译可以加快执行

速度。若脚本具有需要考虑的参数，则最好重用相同的脚本，并为其提供参数

②weight，即重量分数，让函数分数乘以提供的重量作为最终函数得分

③random\_score，使用\_uid字段的散列生成分数，并使用变种种子。 如果未

指定seed，则使用当前时间

④field\_value\_factor，允许用文档中的字段来影响分数。与script\_score

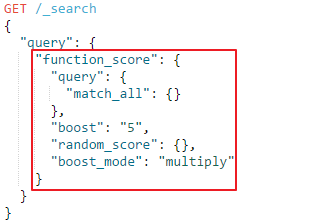
函数类似，但它避免了脚本的开销。如果在多值字段上使用，则仅在计算中使

用该字段的第一个值。例如，假设有一个使用数字字段索引的文档，并希望使

用此字段影响文档的分数。

**例子：**

①function\_score只能与一个函数一起使用：



②function\_score若想与多个函数一起使用，只有当文档与给定的过滤查询匹

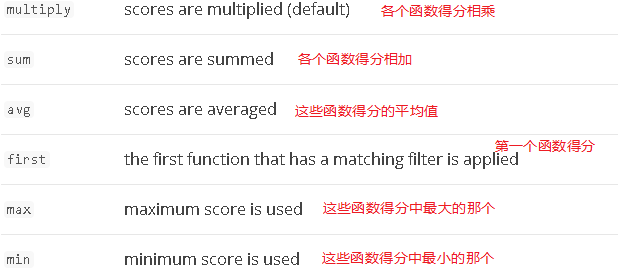
配时才可以，例如：



**额外参数：**

①score\_mode，在function\_score与多个函数使用时用到(即例子②)，它指

定这些函数得分的组合方式，从而得到文档的最终函数得分。取值如下：



②weight，在function\_score与多个函数使用时用到(即例子②)，它指定每

个函数的权重。用它乘以所在函数计算的分数作为此函数的得分

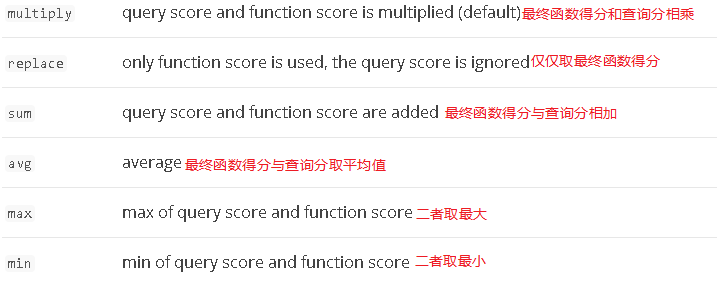
**PS：**如果score\_mode设为avg，则各个分数将通过加权平均值组合。 例如，

两个函数返回得分1和2且它们各自权重是3和4，则它们的得分将组合

为（1 \* 3 + 2 \* 4）/（3 + 4）而不是（1 \* 3 + 2）\* 4）/ 2。

③boost\_mode，定义每个文档的最终函数分数与查询的分数组合方式用来作为

作为文档的\_score值，boost\_mode的组合方式有：



④max\_boost，可以将新分数限制为不超过某个值。它的默认值为FLT\_MAX

⑤min\_score，默认情况下，修改分数不会更改哪些文档匹配。若想排除不满足

特定分数阈值的文档，可以将min\_score参数设置为所需的分数阈值。要使

min\_score起作用，需要对查询返回的所有文档进行评分，然后逐个过滤掉

## 连接查询

像ES这样的分布式系统中执行完整的SQL样式连接查询非常昂贵。 相反地，ES提供两种形式的连接，旨在水平扩展：

①嵌套查询，文档可能包含嵌套类型的字段。这些字段用于索引对象数组，其中

每个对象都可以作为独立文档查询（使用嵌套查询）

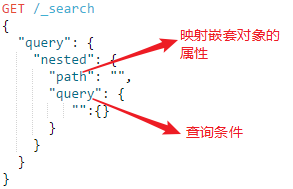
②父子关系嵌套查询，父子关系可以存在于单个索引中的两种文档类型之间。

has\_child查询返回其子文档与指定查询匹配的父文档，而has\_parent查

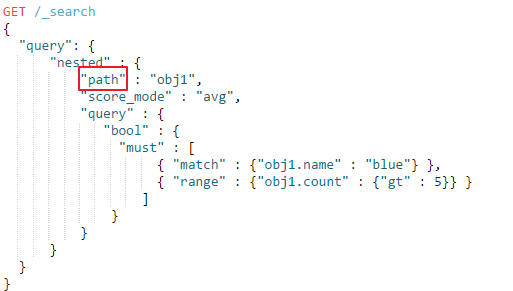
询返回其父文档与指定查询匹配的子文档

### nested

nested query，嵌套查询，用来查询以[nested datatype](#_nested_datatype)映射的文档。**语法：**



**例子：**



**额外参数：**

①score\_mode，允许设置内部对象匹配如何影响父项的评分，默认为avg，

但可以是sum，min，max和none。

②ignore\_unmapped，当设置为true时，将忽略未映射的路径，并且不会匹配

此查询的任何文档。默认值为false，如果未映射路径，查询将引发异常

### has\_child

has\_child接受查询和要运行的子类型，并生成具有与查询匹配的子文档的父文档，[详情查看官网](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-has-child-query.html)。

### has\_parent

has\_parent接受查询和父类型。 查询在父文档空间中执行，该父文档空间由父类型指定。 此查询返回关联父项已匹配的子文档，[官网文档](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-has-parent-query.html)。

### parent\_id

parent\_id可用于查找属于特定父级的子文档

## 地理位置查询

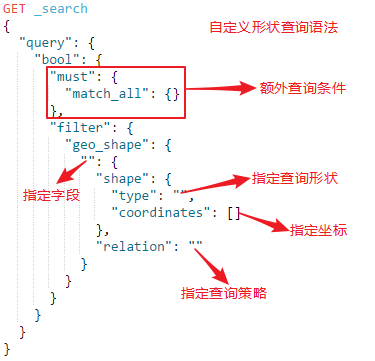
### geo\_shape

geo\_shape query，地理形状查询，用于过滤具有地理形状的文档，这些地理形状要么与指定的地理形状相交，要么包含，要么不与指定的地理形状相交。

geo\_shape query过滤使用geo\_shape datatype索引的文档，它必须在filter(过滤语句)中使用。geo\_shape query使用与geo\_shape datatype相同的网格方形表示来查找具有与查询形状相交的形状的文档，且它还将使用与字段映射定义的相同的PrefixTree配置。

**geo\_shape query支持两种定义查询形状的语法：**

①提供整个形状定义，使用"shape"



②引用一个文档中预先定义好的形状，使用"indexed\_shape"。

这种语法好处是不用每次都去定义坐标，通过index就能引用它。

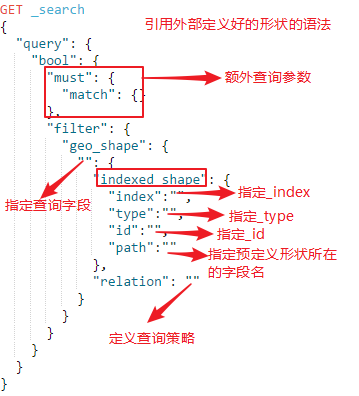
它需要指定4个基本参数：

index：预定义形状所在文档的\_index，默认为shapes

type：预定义形状所在文档的\_type

id：预定义形状所在文档的\_id

path：预定义形状所在的字段名，默认为shape



**额外参数：**

①relation：即查询策略，确定在搜索时可以使用哪些空间关系运算符，

有4个取值:

a.INTERSECTS：匹配与查询形状相交的所有文档(默认值)

b.DISJOINT：匹配与查询形状无任何共同点的所有文档

c.WITHIN：匹配位于查询形状中的所有文档

d.CONTAINS：匹配包含查询形状的所有文档

**例子：**

①自定义边界矩形查询形状，查询位于边界矩形内的所有文档



②自定义圆形查询形状，查询不在此圆形内的所有文档



③引用另一个文档定义好的形状，查询与此形状相交的所有文档，且name字段

值为"addr"



### geo\_bounding\_box

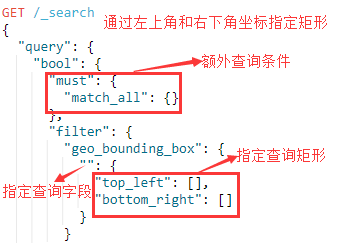
geo\_bounding\_box query，可以用来过滤地理点落在指定矩形内的文档。

它可以处理每个文档的多个位置点， 一旦有一个位置点与其符合，文档就会被匹配到。用于查询数据类型为" geo\_point "的字段且必须放在filter语句中。

geo\_bounding\_box是geo\_shape的一个分支功能，geo\_shape可以通过指定shape.type为"envelope"，就是geo\_bounding\_box的功能。

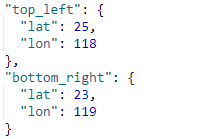
geo\_bounding\_box可以通过两种方式来指定查询参数-矩形：①是指定左上角和右下角地理点；②是指定"上下维度"和"左右经度"。

**语法**：



其中，使用上下左右方式(上面的右图)指定矩形的geo\_bounding\_box，只能用数字表示；对于使用左上角和右下角坐标方式(上面的左图)指定矩形的geo\_bounding\_box，可以有多种方式表示：

①lat(维度)和lon(经度)作为properties



②指定一个Array(注意数组的经纬度表示顺序)



③指定一个字符串(用逗号隔开，前子句表示维度，后子句表示经度)



④指定geohash字符串



**额外参数：**

①\_name：用于标识geo\_bounding\_box的可选名称字段

②validation\_method：有三种取值:

"IGNORE\_MALFORMED"以接受具有无效纬度或经度的地理点；

"COERCE"以尝试推断正确的纬度或经度；

"STRICT"，默认值，只接受正确的经纬度坐标

③type：定义geo\_bounding\_box是在内存中执行还是索引，两种取值：

"indexed"表示在索引中执行；"memory"表示在内存中执行

(默认情况下，geo\_bounding\_box的type设置为memory，这意味着在内存中检查doc是否在边界框范围内。 在某些情况下，索引选项的执行速度会更快。

请注意，在这种情况下，geo\_point类型必须具有lat和lon索引

请注意，使用索引选项时，不支持每个文档字段的多个位置)

**例子：**

1、通过左上角和右下角指定矩形，注意如果用数组[]来表示经纬度，

数组第一个元素表示经度，数组第一个元素才表示维度



2、通过上下左右来指定矩形

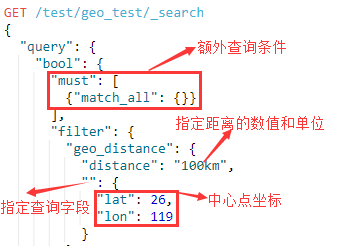


### geo\_distance

geo\_distance query，可以过滤位于指定中心点距离内的地理点所在的文档，它可以处理每个文档的多个地理点，一旦有一个地理点符合geo\_distance，

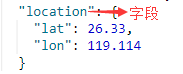
则该文档就会被匹配。同样，geo\_distance也使用在数据类型为geo\_point的字段上且用在filter过滤语句中。使用geo\_distance，需要指定：中心点坐标和距离。

**语法：**



中心点坐标可以有多种表示方式：

①lat属性表示维度，lon属性表示经度



②使用坐标数组，注意坐标数组的经纬度是相反的，它是经度在前纬度在后



③使用坐标字符串，它是纬度在前，经度在后



④使用geohash



**额外参数：**

①distance：指定距离中心点的距离，带单位。请看[ES的距离单位](#_distance_units)

②distance\_type：如何计算距离，有两个取值：arc(弧形)和plane(平面)

默认取值为src；取值为plane时，查询更快，但是在

长距离和靠近极点时不准确。

③\_name：用于标识查询的可选名称字段

④validation\_method：有三种取值：

"IGNORE\_MALFORMED"以接受具有无效纬度或经度的地理点；

"COERCE"以尝试推断正确的纬度或经度；

"STRICT"，默认值，只接受正确的经纬度坐标

**例子：**

1、在location字段中查询地理点位于以中心点(坐标为北纬26，东经119)

100千米的圆形范围内的文档



2、在location字段中查询地理点位于以中心点(坐标为北纬26°，东经119°)

100千米的平面范围内的字段



### geo\_polygon

geo\_polygon query，可以在指定的多边形内过滤包含地理点的文档。

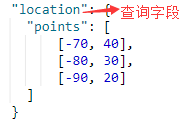
它需要指定一组地理点，通过这组地理点构成一个多边形，来匹配地理点位于这个多边形内的文档。同样用于数据类型为"geo\_point"的字段且必须放在filter过滤语句中使用。

**语法：**

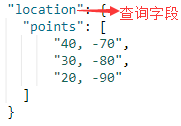


多边形的坐标可以表现成多种格式：

①坐标数组，注意坐标数组的经纬度相反，前一元素是经度，后一元素是维度



②坐标字符串，以逗号隔开，第一个子句是维度，第二个子句是经度



③geohash字符串



**例子：**

1、查询地理点位于指定多边形内的文档

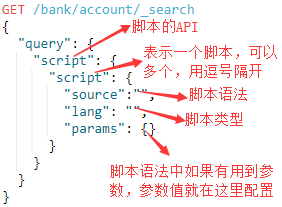


## 脚本查询

### script

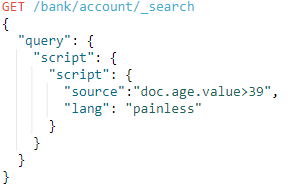
ES在5.x版本后，使用全新的painless脚本，通过脚本可以执行一些DSL难以搭配的复杂查询，详细的painless脚本语法还需要学习。在查询或者过滤子句中，使用“doc”表示文档的内容(即\_source里面我们定义的字段)

**语法：**



**例子：**

①查询age字段大于39的文档

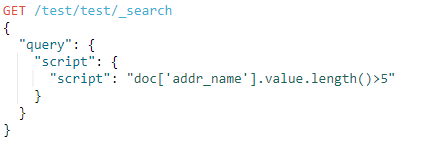


②查询addr\_name字段值的长度大于5的文档，注意：这种需求的查询有两个需要注意的地方：

a.被查询的"text"数据类型的字段，必须设置fielddata=true

b.ES是将字段值分词后拿来查询的，如果要整个字段值的长度拿来比较，需要

设置为keyword，让ES不分词



# ES聚合

ES聚合，全称为"Aggregation"，简称为"aggs"，是ES在索引和检索的基础上提供的更高级的功能：聚合基于搜索查询提供聚合数据，以便可以组合构建复杂的数据。

**ES聚合主要分为4大类：**

①Bucketing，存储桶聚合，与SQL中的"group by"类似。执行聚合时，在上

下文中的每个文档判断存储桶条件，当条件匹配时，文档就会"

落入"该存储桶。整个聚合结束后，会得到一个桶列表，每个桶

内都有一组"属于"它们的文档

②Metric，度量标准聚合，与SQL中的"max"、"min"、"avg"类似。在一组文

档中跟踪和计算指标的聚合

③Matrix，矩阵聚合，在多个字段上运行，并根据从请求的文档字段中提取的

值生成矩阵结果。 与度量标准和存储桶聚合不同，此聚合不支持脚本

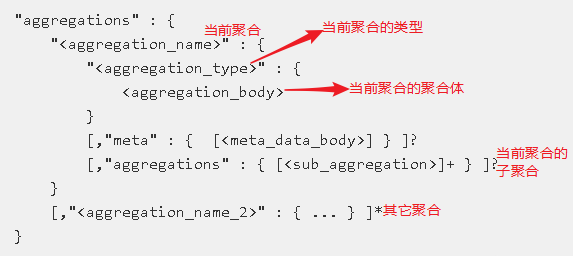
④Pipeline，管道聚合，聚合其他聚合的输出及其关联指标的聚合

**ES聚合的最大特点：**"聚合可以嵌套"：因为每个存储桶定义了一个文档集(属于

该存储桶的所有文档)，所以可以在存储桶级别上关联聚合，

这些聚合将在该存储桶的上下文中执行。

**ES聚合语法结构：**

****

## 度量聚合

metric aggregation，即度量聚合。此类聚合基于某种方式(聚合类型)从当前上下文的文档中提取值来计算度量(度量即最大值、最小值、平均值...)，这些值一般从文档的字段中提取(即使用字段的数据)，也可以使用脚本生成。

### avg，max，min，sum

"avg 聚合"用于从文档集中计算指定字段的平均值；

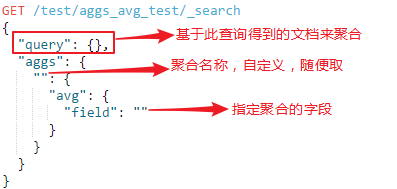
"max 聚合"用于从文档集中找出指定字段的最大值；

"min 聚合"用于从文档集中找出指定字段的最小值；

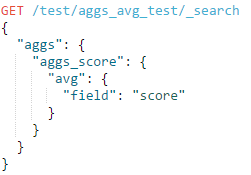
"sum 聚合"用于从文档集中计算指定字段的总和。

以上4个聚合函数用法类似，仅聚合名称不一样而已，以平均值聚合"avg"为例：

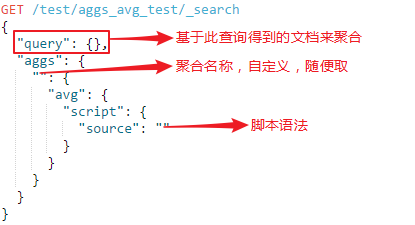
**语法①：通过指定要聚合的字段(field)**



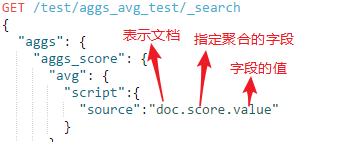
例子为：



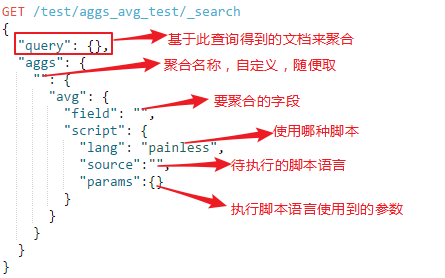
**语法②：指定脚本(script)来执行，source表示待执行的脚本语言**



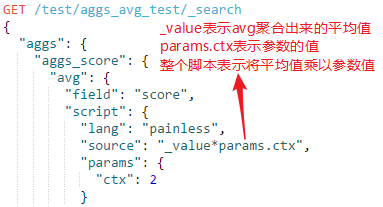
例子为：



**语法③：对聚合出来的平均值再做处理，仍然使用脚本实现**



例子为：



**语法④：对字段没有的值，可以指定另一个值覆盖**



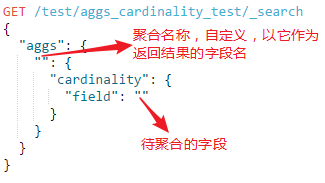
### cardinality

"cardinality 聚合"用于计算文档集中指定字段的值去重后的数量，它将字段的值加载到哈希集中并返回其大小。它有一个额外参数precision\_threshold—用于定义一个唯一的数值，低于此数值的计数接近准确，超过此数值的计数会变得更加模糊。precision\_threshold参数的默认值为3000，最大值为40000(意味着高于40000的数值与40000具有一样的结果)。官方说，可能会在未来版本中更改precision\_threshold选项。

**PS：**若要对[text](#_text_datatype)类型的字段做cardinality聚合，需先设置[fielddata](#_字符串排序)

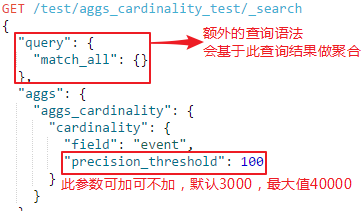
**语法：**

**①指定字段来计算该字段有多少种取值(这些字段值会被去重后统计)**

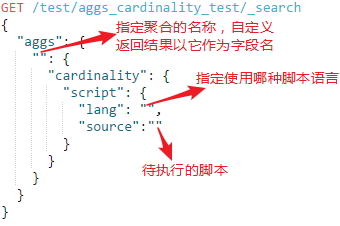


例子为：统计test/aggs\_cardinality/...的文档的event字段的去重取

值数量



**②指定脚本语言来计算字段的去重取值数量**



例子为：

doc表示文档

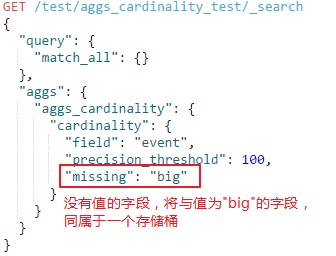
doc["event"]和doc. event效果一样，都是表示文档的event字段

doc.event.value表示文档的event字段的字段值



**③缺少值的字段，默认是被忽略，但是如果有指定missing，则没有值的字段**

**将会与missing指定值的字段属于同一个存储桶**



### stats，extended\_stats

" stats "，用于从文档集指定字段中，提取字段值的统计信息，返回的统计数据包括：min，max，sum，count和avg。所以它只能用于[numeric](#_numeric_datatype)类型的字段。

" extended\_stats "是stats聚合的扩展版本，它在stats原有的基础上添加了额外的度量标准：

①sum\_of\_squares —— 平方和

②variance —— 方差

③std\_deviation——标准差

④std\_deviation\_bounds ——平均值加减两个标准差的值，它有两个属性：

upper和lower。upper=平均值+2\*标准差、lower=平均值-2\*标准差，

ES官方说这用于可视化数据方差，若想要不同的边界，如需要3个标准差，

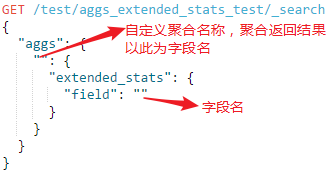
可设置参数sigma为3，则：

upper = 平均值+3\*标准差、lower = 平均值-3\*标准差。

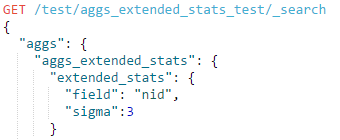
sigma可以是任何非负的双精度，这意味着可以请求非整数值，例如1.5。

**语法：**

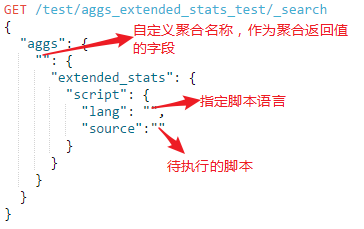
**①通过filed来指定字段，聚合所有文档该字段的数值的统计信息**

****

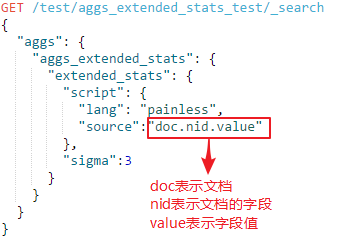
例子为：



**②使用脚本语言来聚合**

****

例子为：

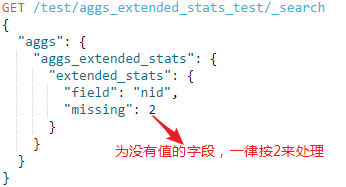
****

**③脚本升级版：对聚合出来的基本数据再做处理**

****

**④对空值的处理**

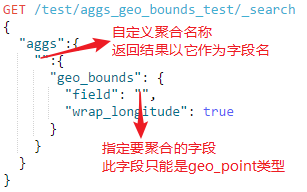
默认情况下，ES会忽略掉空值，但也可以指定



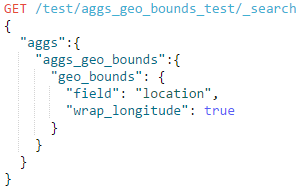
### geo\_bounds

"geo\_bounds 聚合"，用于计算包含文档集中字段所有的geo\_point值的边界框。返回结果是一个用"左上角"坐标和"右下角"坐标表示的矩形边框。该聚合函数还有一个可选参数wrap\_longitude，指定是否允许边界框与国际日期行重叠，默认值是true。

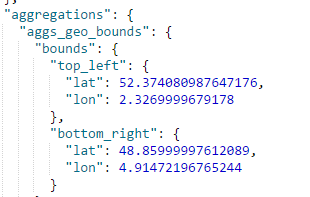
**语法：**



例子为：



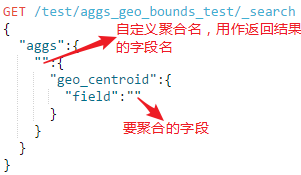
返回结果为：



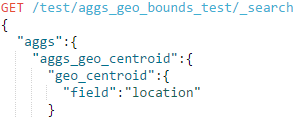
### geo\_centroid

"geo\_centroid 聚合"，计算[geo-point数据类型](#_geo-point_datatype)字段的所有geo\_point值的加权质心。何为质心？即质心算法，质心也是一个坐标，它的横坐标为n个坐标点横坐标的平均值，它的纵坐标为n个坐标点纵坐标的平均值；加权质心，应该是在此基础上加上权值进行计算...吧....（这点有待修改）

**语法：**



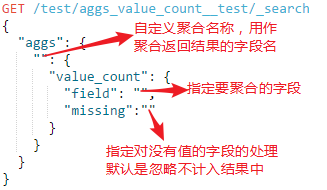
例子为：



### value\_count

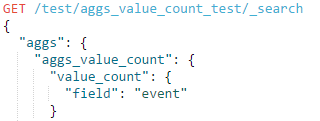
"value\_count 聚合"计算文档集中指定字段的值的数量，它和[cardinality](#_cardinality)聚合的区别在于：value\_count是有一个值算一个，它不会去重，默认忽略没值的字段。换句话说，value\_count是计算文档集中文档的数量。

**语法：**

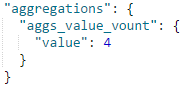


例子为：

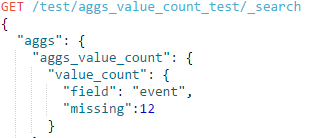
①没有指定对没值字段的处理，默认忽略，不计入统计结果



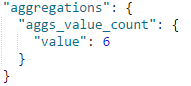
返回值为4：共有6个文档其中2个文档的event字段为null



②若指定对没值字段的处理，则会把没值的字段一个一个计入统计结果



返回值为6：共有6个文档其中2个文档的event字段为null



### top\_hits

"top\_hits 聚合"用于从文档集中返回满足聚合条件的前n个文档。它有3个属性：from、sort和size。而且，它可以让ES聚合结果返回文档信息。

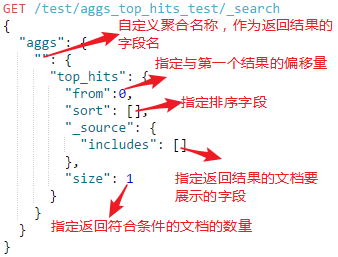
①from：获取相对于第一个结果的偏移量，0表示第一个结果，1表示第二个结

果，3表示第三个结果，以此类推...

②sort：指定排序字段，默认按照主查询的\_score来排序

③size：指定要返回的文档数量，默认前3个匹配的文档

**语法：**



**例子为：**

将文档集按照score字段倒序排序后返回第一个结果，且结果文档只显

示name和score字段



## 桶聚合

Bucket aggregation，即存储桶聚合。它不会像[度量聚合](#_Metric)一样计算字段上的值，而是创建文档桶。每个桶与一个标准(即聚合类型)相关联，该标准确定当前上下文中的文档是否能"落入"其中。

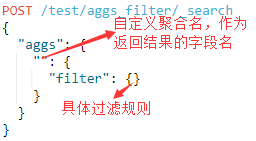
存储桶聚合可以保存子聚合，这些子聚合只对由其父聚合创建的存储桶进行聚合。ES可以嵌套聚合，而且可以多层：父聚合→子聚合→孙聚合...。聚合的顺序是：由外到内聚合，最外层即父聚合先完成，然后在父聚合结果的基础上进行子聚合...以此类推。例如：有公司-部门-产品这种关系，如果对其进行嵌套聚合，顺序为：1、先按照公司划分存储桶；2、在每个公司的存储桶上按照部门划分存储桶；3、在每个公司的每个部门存储桶上按照产品划分文档集。

### filter

filter aggregation，即单桶聚合。过滤当前上下文中与指定过滤器匹配的所有文档，构成一个独立的存储桶，用于将当前聚合上下文缩小到特定文档集。

注意：过滤器使用的语法是[DSL语法](#_DSL语法)，不是聚合语法。

**语法：**



**例子：**

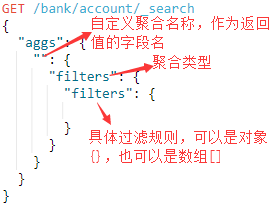
①计算所有T恤类型产品的平均价格



### filters

filters aggregation，即多桶聚合。定义多个存储桶，每个桶与过滤器关联，每个存储桶将收集与其关联过滤器匹配的所有文档。当使用filters聚合且有子聚合时，每个过滤器关联的存储桶都会执行这个子聚合。

**语法：**

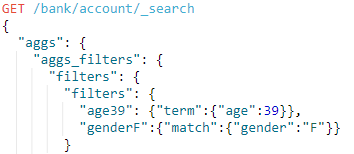


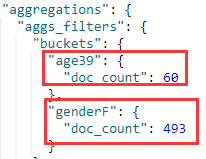
**例子：**

①设置两个存储桶，一个过滤出age=39的文档集，一个过滤出gender=F的文

档集。并且设置第一个存储桶名称(ES称为密钥)为age39，第二个存储桶名

称为genderF。



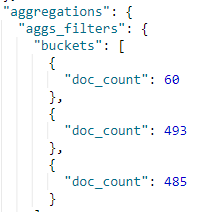
返回结果：  
 

②匿名Filters聚合，不设置桶的名称(ES称为密钥)，用数组形式定义过滤规

则。返回结果按定义规则的顺序返回聚合结果。



返回结果：



**额外参数：**

①other\_bucket：是否聚合与任何给定过滤器都不匹配的所有文档，true为要

聚合。若使用命名过滤器，则该桶默认名称为" \_other\_"，

若使用匿名过滤器，则该桶为返回数组的最后一个桶。

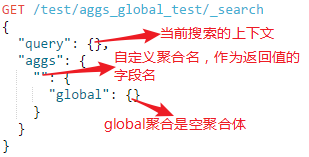
②other\_bucket\_key：修改other\_bucket存储桶的名称，如果使用了这个参

数，则隐式地将other\_bucket设置true。

### global

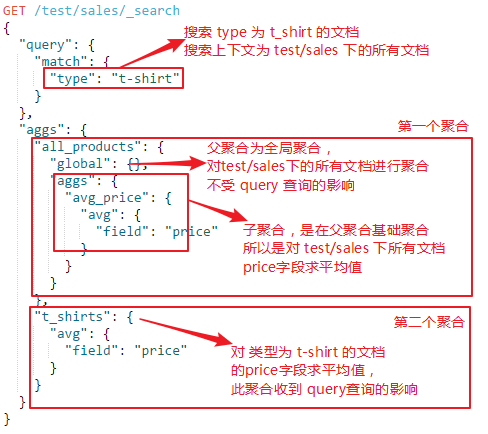
global aggregation，全局聚合。将当前搜索执行上下文中的所有文档作为一个存储桶。该上下文由搜索的index和type定义，但不会受到查询语句本身的影响global aggregation只能作为顶级聚合器放置，因为在另一个桶聚合器中嵌入全局聚合器没有任何意义。

**语法：**



**例子：**

下面的聚合演示了如何在搜索上下文中的所有文档上计算聚合，在下面示例中为第一个聚合的avg\_price，无论查询如何，它将计算目录中所有产品的平均价格，而不仅仅是“衬衫”。而第二个聚合的 t\_shirts 受到 query 查询的影响，它将计算目录中类型为衬衫的平均价格。



### range

range aggregation，即范围聚合。此聚合定义一组范围，每个范围代表一个存储桶。并根据文档指定字段的值判断文档该落入哪一个桶内。请注意：

range聚合包含from值，并排除每个范围的to值。如果range聚合存在子聚合，则每个范围表示的存储桶都会执行子聚合。

**语法：**



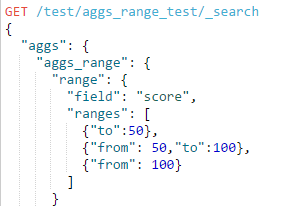
**额外参数：**

①keyed：设置为true，将唯一的字符串键与每个存储桶关联，并将范围作为

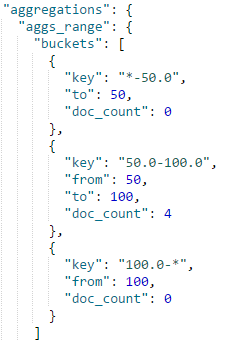
哈希而不是数组返回。

**例子：**

①聚合＜50、50≤x＜100、≤100的文档



返回结果：



②当有子聚合时，每个范围桶都会执行这个子聚合

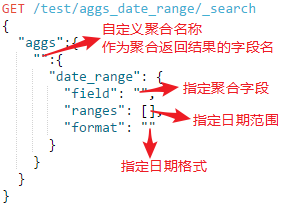


### date\_range

date\_range aggregation，即日期范围聚合。此聚合与正常范围聚合[range](#_range_1)之间的主要区别在于：from和to选项可以使用Date Math表达式，并且可以指定日期格式，通过此日期格式可以返回from和to响应字段。请注意：

date\_range聚合包括from值，但排除每个范围的to值。

**语法：**



**例子：**

①ranges是一个数组，一个对象就是一个聚合，下例中表示2个聚合：

"from": "now-10d"，表示聚合"大于等于当前时间扣去10天的数据"

"to":"now-10d"，表示聚合"小于当前时间扣去10天的数据"

now表示当前时间，10d表示10天，日期表达式可以看[Date Math](#_Date_Math)



返回结果：



**额外参数：**

keyed：设置为true会将唯一的字符串键与每个存储桶关联，并将范围作为哈

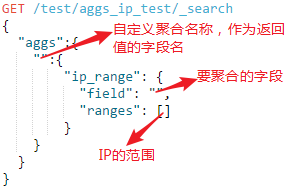
希而不是数组返回。

time\_zone：将日期从另一个时区转换为UTC。

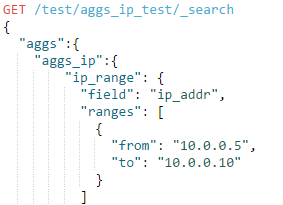
### ip\_range

ip\_range aggregation，IP范围聚合。在[range](#_range_1)聚合的基础上针对IP地址的聚合。同样的，它也是包含from指定的IP地址，去除to指定的IP地址。

**语法：**



**例子：**



### histogram

histogram aggregation，即直方图聚合。基于文档中提取的数值，动态地在值上构建固定大小的存储桶，并将文档向下舍入到其最近的存储桶。

**例如：**

有文档集的值范围是[65,86,95]

如果固定大小设为5，则存储桶为[65,70,75,80,85,90,95]

如果固定大小设为7，则存储桶为[63,70,77,84,91]

**释义：**

...存储桶的取值区间会根据文档集的取值区间，配合用户设置的固定大小来设

置，以上面的例子：固定大小为5，正好5\*13=65，所以存储桶的取值区间

从65开始；固定大小为7，7\*9=63最接近65，所以存储桶的取值区间从63

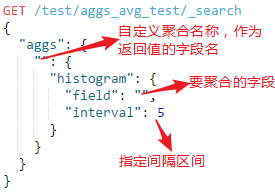
开始。同样，存储桶的取值区间结束也是依据这个理。

...文档是向下舍入到最近的存储桶中，以上面的例子：固定大小为5，存储桶的取值区间为[65,70,75,80,85,90,95]，值65的文档落入值65的存储桶中；值86的文档落入值为85的存储桶中；值95的文档落入值95的存储桶。

也可以这样理解：每个存储桶的取值范围为:65~69,70~74,75~79,80~84,85~

89,90~94,95~99，文档的值位于哪个范围就落入那个存储桶中。

**语法：**



**额外参数：**

①min\_doc\_count：默认情况下，当没有文档落入存储桶，ES将使用空桶填充

直方图中的间隙；如果设置了min\_doc\_count，只会返回

存储桶中文档数大于等于min\_doc\_count的存储桶。

②extended\_bounds：可以“强制”直方图聚合以extended\_bounds.min构

建存储桶，并继续构建存储桶达到最大值。但是，当

extended\_bounds.min高于从文档提取的值区间的最小

值，ES仍是以文档集的最小值构建存储桶，最大值也是

如此。

③order：默认情况下，聚合返回的存储桶按照桶的键值升序排序。可以通过

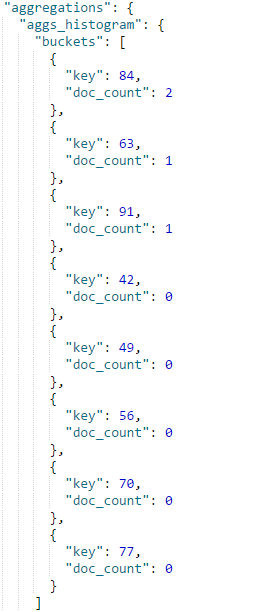
order控制存储桶返回的顺序：\_key:asc，\_count:desc。

\_key表示按照存储桶的键值，\_count表示按照存储桶包含的文档数

**例子：**



返回结果：



### date\_histogram

date\_histogram aggregation，即日期直方图聚合。它以指定字段的最早日期和最晚日期为区间，对其中的日期按照指定间隔划分时间段，并统计落在各个时间段上的文档数量。使用date\_histogram必须指定2个参数：

①field：指定要聚合数据类型为[date](#_date_datatype)的字段

②interval：指定划分的时间间隔，取值有：year、quarter(季度)、month、

week、day、hour、minute、second

额外参数有：

③format：对聚合结果的时间进行格式化，例如"yyyy-MM-dd"

④time\_zone：此参数用于以UTC格式存储日期的字段， 指明应该使用的时区

⑤offset：用于将每个桶的起始时间更改为指定的正(+)或负偏移(-)时间

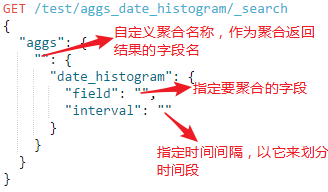
⑥keyed：此参数设为true，会将唯一的字符串键与每个存储桶关联，并将

结果作为哈希返回，而不是数组返回

⑦missing：定义如何处理缺少值的文档，默认是忽略，也可以指定一个值。

所有缺少值的文档都按照这个值来处理。

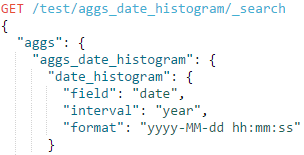
**语法：**



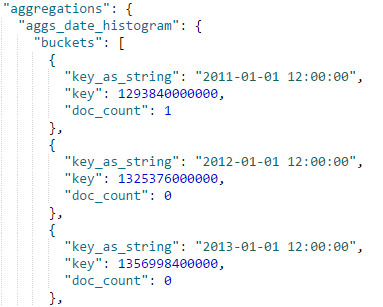
**例子：**

①以"年"为间隔划分时间段，统计每个时间段的文档数量，并将返回结果格式化

注意：它是以字段的最早日期和最晚日期作为时间区间。



返回结果为：



②将keyed设为true，将以哈希返回。使用offset可以将起始时间正偏移或

负偏移一定时间值



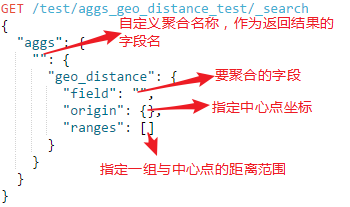
返回结果为：



### geo\_distance

geo\_distance aggregation，即geo范围聚合。通过定义中心点和一组距离范围桶，聚合评估每个文档指定geo\_point字段的值与中心点的距离，根据范围确定该文档所属的桶(如果文档与中心点之间的距离落在桶的距离范围内，则文档属于桶)。

**语法：**



**额外参数：**

①unit：指定距离范围的单位，默认为米(m)，详见[distance units](#_distance_units)

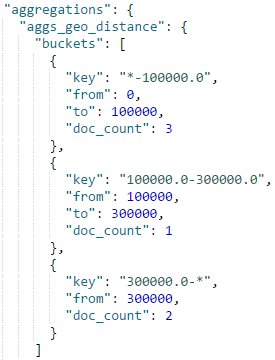
②distance\_type：指定[距离计算模式](#_distance_calculation_modes)，默认为src模式，即圆弧模式

③keyed：将keyed标志设置为true会将唯一的字符串键与每个存储桶关联，

并将范围作为哈希而不是数组返回。

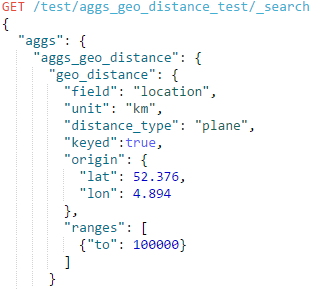
**例子：**

①指定多个与中心点的距离范围

 返回结果：

②指定距离单位，并且指定计算模式，同时将keyed设置为true，但是只

定义一个距离范围



返回结果：



### geohash\_grid

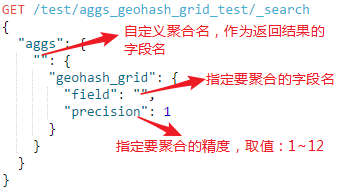
geohash\_grid aggregation，即geo网格聚合。根据文档集中geo\_point字段的经纬度范围，聚合出一个用geohash表示的网格(网格中是一个个基本的单元格)，如下图所示，红点表示落入(1001单元格)中：



ES聚合出来的网格也类似上面的例子，将文档集中的geo\_point字段的经纬度值按照精度参数precision聚合出一个网格，网格由一个个单元格组成，每一个单元格为一个存储桶，统计落在存储桶(单元格)中的文档数量，存储桶的唯一标识是单元格表示的geohash值。

**语法：**

precision值越大，精度越高，单元格越多，单元格表示的geohash值越大。它的取值范围为：1~12，值为12的最高精度geohash产生的单元覆盖不到一平方米的土地。



**额外参数：**

①size：要返回的最大geohash存储桶数（默认为10000）。根据包含的文档量

对存储桶进行优先级排序。

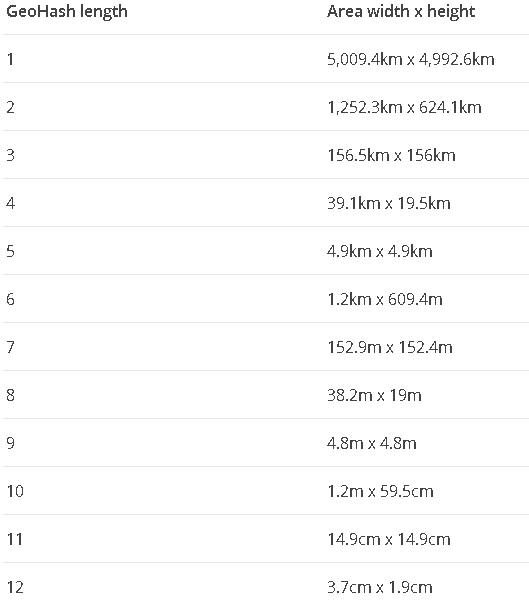
②shard\_size：设置每个分片需要返回的存储桶，默认从每个分片返回

max（10，（size x of-shards））个存储桶。

**补充：**

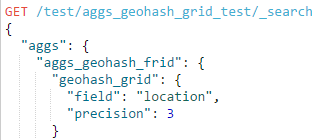
下表显示了各种字符串长度的geohash所涵盖的单元格的度量标准维度；

单元尺寸随纬度而变化，因此该表适用于赤道最坏的情况：

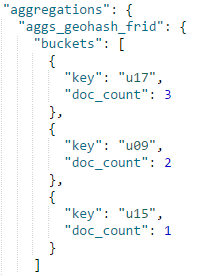


**例子：**

①精度小的geohash\_grid聚合，可以直接使用



返回结果：每个存储桶的Key为聚合单元格的geohash值



②精度大的geohash\_grid聚合，由于ES是将指定文档集中所有的geo\_point

字段拿去聚合成一个网格，这些字段间的经纬度位置差距会导致返回的单元格

(即存储桶)很多，甚至达到百万级别。因此先用[Filter聚合](#_filter)和[geo过滤器](#_geo_bounding_box)缩

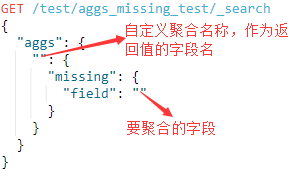
小搜索的范围：



### missing

missing aggregation，即缺值聚合。该聚合创建当前上下文中缺少字段值（缺少字段或具有配置的NULL值）的所有文档的存储桶。通常与其它字段数据聚合（例如range aggregation）结合使用，用来返回由于缺少字段数据值而无法放置在任何其他存储区中的所有文档的信息。

**语法：**



### nested

nested aggregation，即嵌套文档聚合。该聚合只能用于字段类型是[nested dataType](#_Nested_datatype)的文档集，创建一个存储桶，将嵌套文档放入该桶内。对这个存储桶作子聚合，就可以在这些嵌套文档上定义任何类型的聚合。

**PS：**nested aggregation的子聚合在使用嵌套文档的字段时，需要结合顶级

文档中的字段，如例如①所示。

**语法：**



**例子：**

①计算产品（product）的最小价格（price），其中价格是作为嵌套文档保存到

产品中

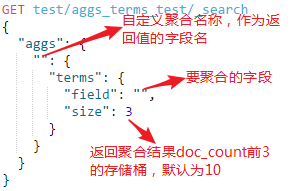


### terms

terms aggregation，即术语聚合。根据字段的值构建存储桶，每个唯一值构建一个存储桶。当ES存在多个主分片时，协调节点会从每个分片上请求提供其自己的最大大小的术语桶，这将意味着返回的列表略有偏差且不准确，详见

[terms聚合的不确定性](#_terms聚合的不确定性)。

**语法：**



**额外参数：**

①size：设置size参数以定义应从整个术语列表中返回多少个术语桶。

②shard size：设置协调节点将从每个分片请求多少个术语桶。该参数的大小

应该要大于参数size的大小（不然没意义，分片只返回3个，

但是结果需要5个，协调节点无法整合）。shard size请求的

大小越高，结果越准确，但计算结果更高。

③order：设置返回结果的存储桶的顺序，默认情况下，存储桶按其doc\_count

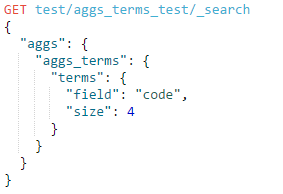
降序排序。(doc\_count是聚合后落入该桶的文档数量)

④min\_doc\_count：设置仅返回超过min\_doc\_count指定数量的术语存储桶。

**例子：**

①对code字段的值进行术语整合，每种值作为一个存储桶，按doc\_count排

序只返回前4个桶



返回结果：

doc\_count\_error\_upper\_bound：每个术语的文档错误的上限。

sum\_other\_doc\_count：当有很多独特的术语时，ES只返回顶部术语，此字段

是不属于响应的所有存储桶的文档计数总和。



# ES补充点

## ES日期表达式

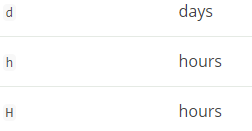
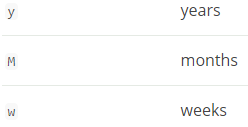
ES的[date数据类型](#_date_datatype)支持日期表达式，即Date Math，表达式以锚定日期开始，可以是现在，也可以是以||结尾的日期字符串。该锚定日期可以可选地后跟一个或多个数学表达式，例如：

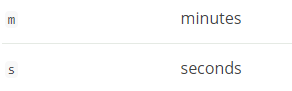
"now+1h"：表示在当前时间上加一小时

"now-1d"：表示在当前时间上减一天、

"/d"：表示四舍五入到最近的一天

ES支持的时间单位有：

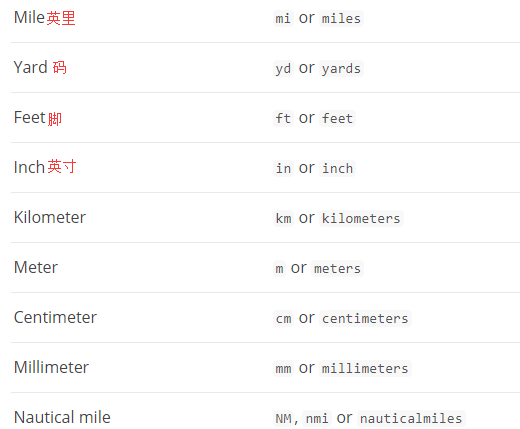




## ES距离单位

ES的距离单位，即distance units。无论何处需要指定距离，例如地理距离查询中的距离参数，如果未指定任何距离，则默认单位为m(米)。 距离可以在其他单位指定，例如“1km”或“2mi”（2英里）。[详情查看官网](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/common-options.html#distance-units)。

常用的距离单位如下图所示：



## ES距离计算模式

ES距离计算模式，即distance calculation modes。在ES中，对地理位置距离查询和聚合，有两种距离计算模式：arc(弧)和plane(平面)，默认都以arc计算距离。

**区别：**

①arc：最准确但速度一般

②plane：计算速度最快但不准确

**使用场景：**

①当搜索环境"狭窄"，跨越较小地理区域(5km以内)，推荐使用plane模式

②对于跨越非常大的区域(如，跨大陆搜索)的搜索，使用plane模式将返回

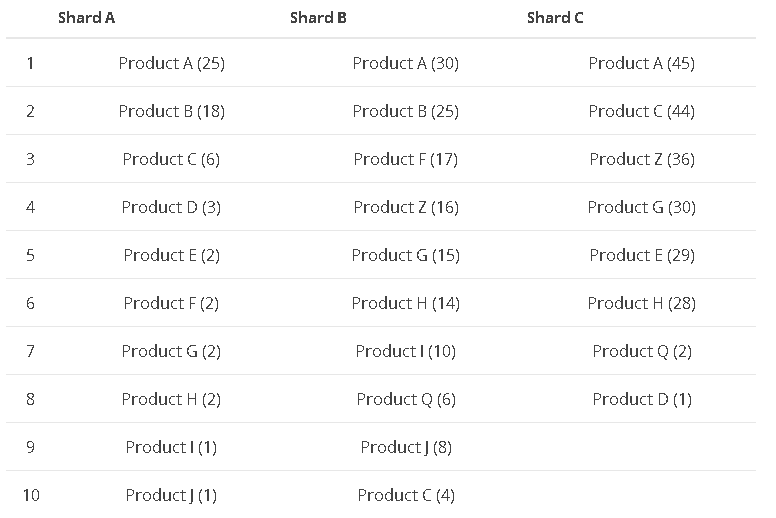
更高的误差范围，推荐使用arc模式

## terms聚合的不确定性

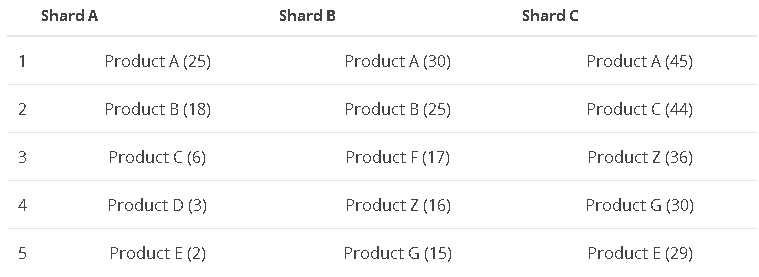
术语聚合( terms aggregation)的文档计数（以及任何子聚合的结果）并不总是准确的。 这是因为每个分片都提供了自己的有序列表应该是什么的视图，并将它们组合在一起以给出最终视图。例如：存在这种情况：

①请求获取产品中的前5个术语，按照具有3个分片的索引的降序文档计数排序。 在这种情况下，要求每个分片给出前5个术语。

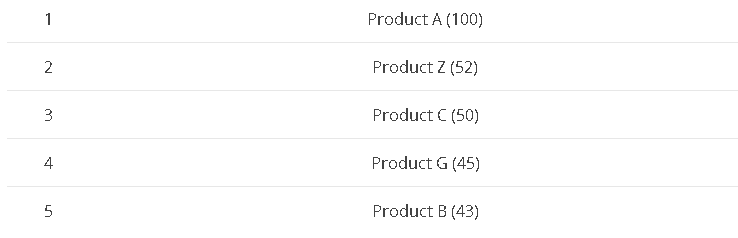
三个分片中的每个分片的术语如下所示，其各自的文档计数用括号表示：



分片将返回它们的前5个术语，因此3个分片的结果将是：



协调节点将从每个分片中获取前5个结果（根据要求）并将它们组合成最终的前5个列表会产生以下结果：



因为产品A是从所有分片返回的，所以我们知道它的文档计数值是准确的。 产品C仅由分片A和C返回，因此其文档计数显示为50，但这不是准确的计数。 产品C存在于分片B上，但其数量不足以将产品C放入该分片的前5个列表中。 产品Z也只返回2个分片，但第三个分片不包含该术语。 在将结果组合在一起以产生最终术语列表时，无法知道产品C的文档计数存在错误，而产品Z的文档计数存在错误。产品H的文档数量为44 ，但是3个分片都没有将其包含在最终的术语列表中，因为它没有进入任何分片的前五个术语。

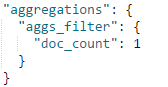
## ES聚合返回文档信息

ES聚合只统计符合聚合条件的文档的数量，并没有返回具体的文档信息，如果需求中需要知道聚合后的文档信息，可以借助[top\_hits](#_top_hits)聚合来返回，将它作为子聚合即可：

①若没有使用top\_hits聚合，下面的聚合只会返回文档数量



返回结果：



②必须为aggs\_filter聚合添加一个子聚合[top\_hits](#_top_hits)，让它会在父聚合的基

础上再做聚合，并且结合[top\_hits](#_top_hits)自身的特性，将父聚合上的文档返回。

使用top\_hits时，最好是指定sort和size，让文档有顺序的返回，当然如

果父聚合后的文档没有任何顺序，那只要指定size即可。



返回结果：

